May 1995

UILU-ENG-95-2215
CRHC-95-11

Center for Reliable and High Performance Computing

Software Dependability in the Operational Phase

Inhwan Lee

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois	6b. OFFICE SYMBOL (If applicable) N/A
7a. NAME OF MONITORING ORGANIZATION NASA ONR	
6c. ADDRESS (City, State, and ZIP Code) 1308 W. Main St. Urbana, IL 61801	7b. ADDRESS (City, State, and ZIP Code) NASA Langley Res. Center, Hampton, VA 23665 Office of Naval Res., 800 N. Quincy, Arlington, VA 22217
8a. NAME OF FUNDING/SPONSORING ORGANIZATION NASA, ONR, Tandem Computers	8b. OFFICE SYMBOL (If applicable)
9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER NASA NAG-1-613 N00014-91-J-1116	
8c. ADDRESS (City, State, and ZIP Code) See 7b. Tandem Computers, Cupertino, CA	10. SOURCE OF FUNDING NUMBERS

PROGRAM ELEMENT NO.	PROJECT NO.	TASK NO.	WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)
Software Dependability in the Operational Phase

12. PERSONAL AUTHOR(S) Inhwan Lee

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM ______ TO ______	May 1995	110

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB-GROUP

18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
dependability, measurement, software faults, error propagation

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Software quality should be built-in and maintained throughout the software life cycle, which requires understanding of software dependability in actual environments. This thesis discusses how to develop analysis techniques for evaluating the dependability of operational software using real measurements while taking design issues into account. The issues addressed include fault categorization and characterization of error propagation, symptom-based diagnosis of recurrent software failures, identification of software fault tolerance, evaluation of the impact of software faults on the overall system, and the development of techniques for analyzing multiway failure dependencies among software and hardware modules. The process is illustrated using a case study of the Tandem GUARDIAN operating system. (OVER)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD FORM 1473, 84 MAR — 83 APR edition may be used until exhausted. All other editions are obsolete.

SECURITY CLASSIFICATION OF THIS PAGE
UNCLASSIFIED

Abstract Cont.

Using process pairs in Tandem systems, which was originally intended for tolerating hardware faults, allows the system to tolerate about 70% of reported faults in the system software that cause processor failures. The loose coupling between processors, which results in the backup execution (the processor state and the sequence of events) being different from the original execution, is a major reason for the measured software fault tolerance. About 72% of reported field software failures in Tandem systems are recurrences of previously reported faults. In addition to the conventional approach of reducing the number of faults in software, software dependability in Tandem systems can be enhanced by reducing the recurrence rate and by improving the robustness of process pairs and the system configuration. An approach for automatically diagnosing recurrences based on their symptoms is developed. The results of evaluations of the effectiveness of the approach show that between 75% and 95% of recurrences can be successfully identified by matching failure symptoms, such as stack traces and problem detection locations. Less than 10% of faults are misdiagnosed.

SOFTWARE DEPENDABILITY IN THE OPERATIONAL PHASE

BY

INHWAN LEE

B.S., Seoul National University, 1979
M.S., Seoul National University, 1985

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1994

Urbana, Illinois

SOFTWARE DEPENDABILITY IN THE OPERATIONAL PHASE

Inhwan Lee, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1994
Ravishankar K. Iyer, Advisor

Software quality should be built-in and maintained throughout the software life cycle, which requires understanding of software dependability in actual environments. This thesis discusses how to develop analysis techniques for evaluating the dependability of operational software using real measurements while taking design issues into account. The issues addressed include fault categorization and characterization of error propagation, symptom-based diagnosis of recurrent software failures, identification of software fault tolerance, evaluation of the impact of software faults on the overall system, and the development of techniques for analyzing multiway failure dependencies among software and hardware modules. The process is illustrated using a case study of the Tandem GUARDIAN operating system.

Using process pairs in Tandem systems, which was originally intended for tolerating hardware faults, allows the system to tolerate about 70% of reported faults in the system software that cause processor failures. The loose coupling between processors, which results in the backup execution (the processor state and the sequence of events) being different from the original execution, is a major reason for the measured software fault tolerance. About 72% of reported field software failures in Tandem systems are recurrences of previously reported faults. In addition to the conventional approach of reducing the number of faults in software, software dependability in Tandem systems can be enhanced by reducing the recurrence rate and by improving the robustness of process pairs and the system configuration. An approach for automatically diagnosing recurrences based on their symptoms is developed. The results of evaluations of the effectiveness of the approach show that between 75% and 95% of recurrences can be successfully identified by matching failure symptoms, such as stack traces and problem detection locations. Less than 10% of faults are misdiagnosed.

Acknowledgments

I would like to express deep gratitude to my thesis advisor, Professor Ravi Iyer, for his guidance, encouragement, and support throughout this thesis work. I would also like to thank Professor Sung Mo Kang for advising me during the first two years of my Ph.D. study, Professor Daniel P. Siewiorek at the Carnegie Mellon University for his thoughtful comments on this research, and Professors Prithviraj Banerjee, Roy Campbell, W. Kent Fuchs, and Benjamin W. Wah for serving on my dissertation committee.

Thanks are due to my colleagues at the Center for Reliable and High-Performance Computing for creating a stimulating environment and offering their friendship. Gwan Choi, Robert Dimpsey, Kumar Goswami, Wei-Lun Kao, Greg Ries, Dong Tang, Tim Tsai, and Steve VanderLeest provided valuable comments on this research. Hungse Cha and Sungho Kim gave me many motivational talks during the final stretch. Fran Wagner kindly proofread this thesis. I appreciate all of their efforts and encouragement. Special thanks are due to Tandem Computers Incorporated, and especially to Bob Horst, Abhay Mehta, and Gil Pitt for their assistance with my thesis research.

Finally, I would like to thank my parents, my brother and three sisters for their love and understanding.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

While hardware reliability has improved significantly in recent decades, improvements in software reliability have not been as pronounced. Hardware faults are generally well understood, efficient hardware fault models exist, and hardware fault tolerance is relatively inexpensive to implement. Unfortunately, the same is not true for software. Software faults are logically complex, poorly understood, and hard to model. Software fault tolerance is a moving target: software quality keeps changing with design updates, and software varies significantly from system to system. To further complicate matters, software interacts with hardware and the environment, blurring the boundaries between them. It is generally believed that software is the major source of system outages in fault-tolerant systems [1], [2]. Given the increasing size and complexity of software, this trend is likely to continue.

Software quality should be built-in and maintained throughout the software life cycle. This requires understanding software dependability in actual environments. There is no better way to understand the dependability characteristics of a complex software system than through direct measurement and analysis. This thesis discusses development of analysis techniques for evaluating the dependability of operational software using real measurements while taking design issues into account. In this thesis, measurements mean the monitoring and recording of naturally occurring errors and failures in the running system under user workloads.

Many studies have sought to improve the software development process by using the failure data collected during the development phase. A typical assumption was that the development process is the only important factor determining software dependability. However, dependability issues for operational software can be very different from those for the software during its development, due to differences in the operational environment and software maturity. During the operational phase, software is not an independent entity but a part of the system. Thus, the dependability of operational software has to be investigated in the context of the overall system. Although the dependability of software is eventually judged by the software's behavior in the operational phase, dependability evaluation of operational software has been a greatly neglected area. This research fills the vacuum by bringing practical issues in designing and maintaining large software systems together with theoretical issues, such as problem diagnosis, fault tolerance, and modeling and analysis.

A study of the dependability of operational software based on real measurements is not simply an analysis of data. It involves instrumentation and requires understanding system architecture, hardware, and software as well as the development, service, and operational environments. A major question is: what can we say about future systems based on measurements from current systems? Another major question is: what can we say about general designs based on measurements of a specific design? A study based on real measurements may not address all design areas. In some areas, however, it can provide very important understanding of how well current techniques work and can identify the critical dependability issues that must be addressed. This thesis identifies and addresses these areas based on measurements taken from a fault-tolerant software system: the Tandem GUARDIAN operating system.

This research consists of two major parts: analysis and design. The analysis covers software fault categorization and characterization of software error propagation, identification of software fault tolerance, evaluation of the impact of software faults on the overall system, and the development of techniques for analyzing multiway failure dependencies among software and hardware modules. The results of the analysis are used for design. Based on the analysis results, this thesis develops a system-independent approach for automatically diagnosing recurrences based on symptoms.

1.1 Related Research

Software errors in the development phase have been studied by researchers in the software engineering field. Software error data collected from the DOS/VS operating system during the testing phase was analyzed in [3]. A wide-ranging analysis of software error data collected during the development phase was reported in [4]. An error analysis technique was used to evaluate software development methodologies in [5]. Relationships between the frequency and distribution of errors during software development, maintenance of the developed software, and a variety of environmental factors were analyzed in [6]. The orthogonal defect classification, an approach to use observed software defects to provide feedback on the development process, was proposed in [7]. These studies mainly attempt to fine-tune the software development process based on error analysis.

Software reliability modeling has been studied extensively, and many models have been proposed [8], [9]. For the most part, these models attempt to estimate the reliability of software by analyzing the failure history of software during the development phase, verification efforts, and operational profile.

Measurement-based analysis of operational software dependability has evolved over the past 15 years. An early study proposed a workload-dependent probabilistic model for predicting software errors based on measurements from a DEC system [10]. A study of failures and recovery of the MVS operating system running on an IBM 3081 machine addressed the issue of hardware-related software errors [11]. The effect of workload on operating system reliability has been analyzed using the data collected from an IBM 3081 running VM/SP [12]. A Markov model that describes the software error and recovery process in a production environment using error logs from the MVS operating system was discussed in [13]. A recent analysis of data from the IBM/MVS system investigated software defects and their impact on system availability [2]. A census of Tandem system availability has shown that, as the reliability of hardware and maintenance improves, software becomes the major source (62%) of outages in the Tandem system [1].

Symptoms of faults in computer systems have been studied using error logs. An information organization and data reduction concept for fault prediction, *tuple*, was developed in [14]. Separation of an error log into transient and intermittent events, and failure prediction based on the shape of the interarrival time function were discussed in [15]. A probabilistic methodology for recognizing the symptoms of persistent problems was proposed and illustrated using error data collected from an IBM 3081 and two CYBER systems [16].

Failure diagnosis has attempted to locate the underlying faults of failures. Symptom-directed diagnosis of system faults was discussed in [17], [18]. An expert system to help analyze crashes of the VMS operating system using the crash dump files and system event logs as data was presented in [19]. Detection and discrimination of network faults based on network traffic signatures were studied in [20]. The *recreate* problem in identifying and diagnosing software failures in the field was discussed in [21].

Software failures have also been studied from the software fault tolerance perspective. Two major approaches for software fault tolerance—recovery blocks and N-version programming—were proposed in [22], [23]. Dependability modeling and evaluation of these two approaches were discussed in [24]. An approximate model to account for failures due to design faults was derived and used to evaluate a fault-tolerant software system in [25]. The effectiveness of recovery routines in the MVS operating systems was evaluated using measurements from an IBM 3081 machine in [26]. Software fault tolerance in the Tandem GUARDIAN operating system was discussed in [27], [28]. Architectural issues for incorporating hardware and software fault tolerance were discussed in [29], [30], [31].

1.2 Contributions

The contribution of this thesis is that it identifies and addresses critical dependability issues for large, continually evolving, operational software. Using measurements collected from the Tandem GUARDIAN operating system, this thesis demonstrates how to develop analysis techniques for evaluating the dependability of operational software while taking design issues into account. This research consists of two major parts: analysis and design. The analysis covers software fault categorization and characterization of software error propagation, identification of software fault tolerance of process pairs, evaluation of the impact of software faults on the overall system, and the development of techniques for analyzing multiway failure dependencies among software and hardware modules. Based on the results of analysis, this thesis develops a system-independent approach for automatically diagnosing recurrences based on their symptoms. The numerical results are specific to the measurements, but the methods and principles apply to other studies.

1.2.1 Analysis

This research found that about 72% of reported field software failures in Tandem systems are recurrences of previously reported faults. This result shows that, in environments where many users run the same software, the number of faults in software is not the only important factor. Recurrences can seriously degrade software dependability in the field. The investigation of failure symptoms showed that failures caused by the same software fault often have identical stack traces, which suggests that automatic diagnosis of recurrences based on symptoms might be possible. Further analysis showed that error propagation and modular program structure are major reasons that failures caused by the same software fault have different stack traces. Consistency checks made by the operating system help failures caused by the same software fault have identical stack traces by preventing error propagation.

The results showed that hardware fault tolerance buys software fault tolerance. Using process pairs in Tandem systems, which was originally intended for tolerating hardware faults, allows the system to tolerate about 70% of reported faults in the system software that cause processor failures. The loose coupling between processors, which results in the backup execution (the processor state and the sequence of events occurring) being different from the original execution, is a major reason for the measured software fault tolerance. The results indicated that the actual level of software fault tolerance achieved by the use of process pairs depends on the degree of difference in the processing environment between the original and backup executions and on the proportion of subtle faults in the software. While process pairs may not provide perfect software fault tolerance, the implementation of process pairs is not as prohibitively expensive as is developing and maintaining multiple versions of large software programs.

Missing operations and not providing routines to handle rare but legitimate operational scenarios are the most common types of software faults in Tandem systems. The data showed that there is a 60% chance that a single program variable acquires an initial, incorrect value when software faults are exercised. In about 20% of the cases, multiple program variables are affected simultaneously. Once errors are generated, the three major error propagation modes are: the first error is certain to be detected on the first access by consistency checks (no propagation, 31%); the problem is detected shortly after the first error is accessed and used (quick detection, 39%); and the first error causes more errors, which are detected after a significant latency (further corruption, 18%). In about half of the failures, problems are detected by consistency checks; in the other half, problems are detected as a result of address violations.

This research developed a method for analyzing multiway failure dependencies among software and hardware modules. The method is based on multivariate statistical techniques, such as factor analysis and cluster analysis. An illustration of the method using processor halt logs demonstrated that factor analysis can unearth the underlying multiway failure dependencies and that cluster analysis can identify the actual dependency patterns.

In addition to the conventional approach of reducing the number of faults in software, software dependability in Tandem systems can be enhanced by reducing the recurrence rate and by improving the robustness of process pairs and the system configuration. The number of faults in software and the recurrence rate are general factors; the robustness of process pairs and the system configuration are platform-dependent factors. The impact of software fault tolerance (i.e., the robustness of process pairs) and the impact of system configuration are as significant as is the impact of the number of faults in software. A complete elimination of recurrences would triple the mean time between software failures.

1.2.2 Design

After all analyses were finished, one issue stood out. This issue is recurrence. This research developed a system-independent approach for automatically diagnosing recurrences based on symptoms, for use in environments where many users run the same software. Specifically, we proposed the comparison of stack traces and problem detection locations as a strategy for identifying recurrences. We applied this strategy using failures in two Tandem system software products and compared the results obtained with actual diagnosis and repair logs from analysts.

The comparison showed that between 75% and 95% of recurrences can be successfully identified by matching failure symptoms, such as stack traces and problem detection locations. Less than 10% of faults are misdiagnosed. The results show that the proposed automatic diagnosis of recurrences allows analysts to diagnose only one out of every several software failures (i.e., primarily the failures caused by new faults). In the case of a recurrence for which the underlying cause was identified, the diagnosis tool can rapidly provide a solution. In the case of a recurrence for which the underlying cause is being investigated, the diagnosis tool can prevent a repeated diagnosis by identifying previous failures caused by the same fault. These benefits are not free of cost. Misdiagnosis is harmful, because a single misdiagnosis can result in multiple additional failures. (Such a danger exists in diagnoses by analysts, also.) The proposed approach needs to be implemented in a pilot. Measurements need to be made to determine how well the approach works and to make design trade-offs.

1.3 Overview

Chapter 2 introduces the Tandem GUARDIAN system and measurements. Two types of measurements are described: human-generated software failure reports and on-line event logs automatically generated by the operating system.

Chapters 3 through 6 discuss the analysis part of this work. Chapter 3 categorizes the underlying faults of software failures, identifies the immediate effects of the faults on the processor state, and traces the propagation of the effects on other system areas until problems are detected by the operating system. The issue of recurrence is also discussed.

Chapter 4 evaluates the software fault tolerance of process pairs in the Tandem GUARDIAN system. Two types of analyses are performed. First, the level of software fault tolerance achieved by the use of process pairs and the detailed reasons for software fault tolerance are investigated, using human-generated software failure reports. Next, the impact of software failures on system performance and the effectiveness of the built-in single-failure tolerance of the Tandem system against software failures are evaluated by conducting Markov reward analysis, using on-line processor halt logs.

Chapter 5 presents a method for analyzing multiway failure dependencies among software and hardware modules. The method is based on multivariate statistical techniques, such as factor analysis and cluster analysis. The method is illustrated using on-line processor halt logs.

Chapter 6 identifies the factors which determine the dependability of operational software. The chapter builds a model to describe the impact of software faults on an overall Tandem system in the field. The model is used to evaluate the significance of the factors considered and to identify areas where improvement efforts can be directed by conducting sensitivity analysis.

Chapter 7 discusses a design based on the analysis results. The chapter develops a system-independent approach for automatically diagnosing recurrences based on their symptoms. A diagnosis environment is discussed, a diagnosis strategy (i.e., a set of symptoms and an associated matching scheme for diagnosing recurrences) is proposed, and a method for evaluating the effectiveness of a diagnosis strategy is presented. The effectiveness of the proposed diagnosis strategy is evaluated using actual failure and repair data collected from two Tandem system software products. Chapter 8 concludes this research.

Chapter 2

Tandem System and Measurements

The Tandem GUARDIAN system is a message-based multiprocessor system built for on-line transaction processing. Fault tolerance is a primary design objective. The design approach is to control fail-fast hardware modules using fault-tolerant software with little wasted redundancy [28]. A Tandem GUARDIAN system of the type studied here consists of 2 to 16 processors, dual interprocessor buses, dual-port device controllers, input/output devices, multiple I/O buses, and redundant power supplies.

In the Tandem GUARDIAN system, a critical system function or user application is replicated on two processors as primary and backup processes, i.e., as process pairs. Normally, only the primary process provides service. The primary sends checkpoints to the backup, so that the backup can take over the function when the primary fails. The GUARDIAN system software halts the processor it runs on when it detects nonrecoverable errors. The "I'm alive" message protocol allows the other processors to detect the halt and to take over the primaries that were running on the halted processor. Redundancy in the processor-device interconnect is provided by allowing each processor to access an input/output device through a dual-port device controller. Redundancy for the disk-resident database is provided by disk mirroring. This approach keeps two physical copies of a database on different disks, which are accessible through a pair of dual-port disk controllers, hence providing eight paths to the database.

Figure 2.1 illustrates the software failure and recovery process in the Tandem GUARDIAN system. When a fault in the system software is exercised, an error (a first error) is generated. Depending on the processor state, this error may disappear or cause additional errors before being detected. The impact of a detected error ranges from a minor cosmetic problem at the user/system interface to a database corruption. A software failure occurs when the system software detects nonrecoverable errors and asserts a processor halt. Once a software failure occurs, the system attempts to recover from the failure using backup processes on other processors. If this recovery is successful, the system can, without noticeable degradation, tolerate the software fault that caused the halt. If a job takeover is not successful, or if a backup process faces the same problem after a takeover, a double processor halt occurs. Whether or not the recovery is successful, the software fault is identified and a fix is made. A single software fault can cause multiple software failures at a single site or at multiple sites.

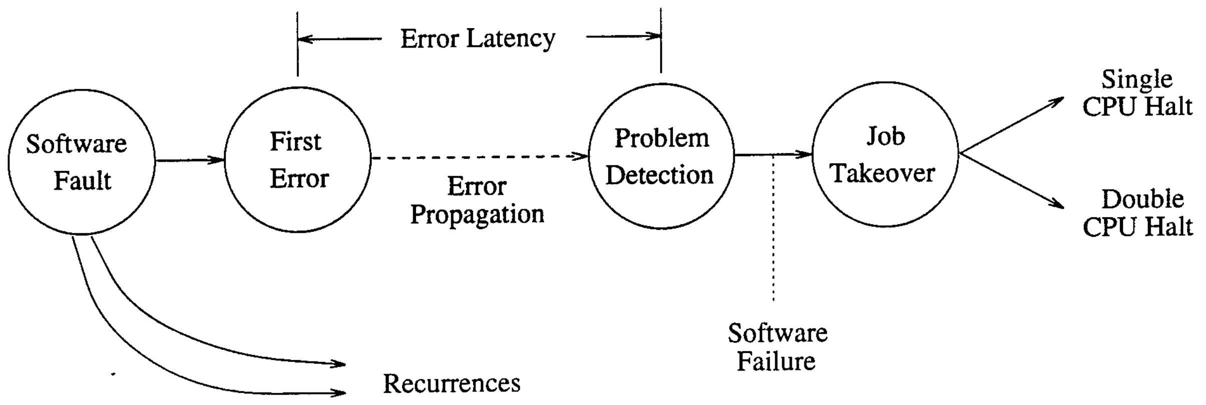
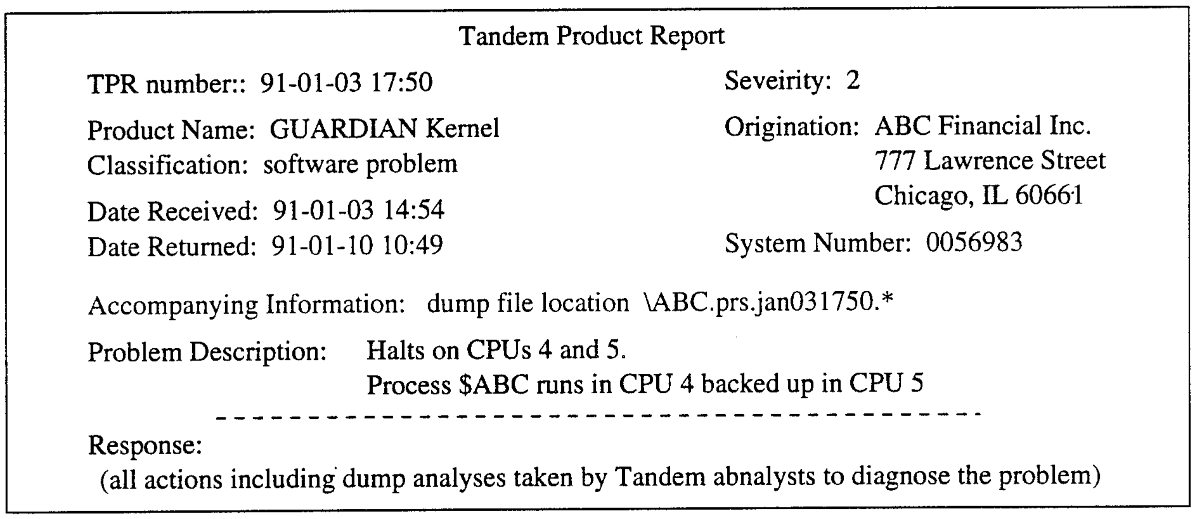


Figure 2.1 Software Failure and Recovery in the Tandem GUARDIAN System

This research focuses on a class of faults and errors that cause software failures. Two types of data were used: human-generated software failure reports, and on-line processor halt logs automatically generated by the operating system. Human-generated software failure reports provide detailed information about the underlying faults, failure symptoms, and fixes. On-line processor halt logs provide close to 100% of reporting and accurate timing information on processor failure and recovery. Both types of data are essential for believable dependability analysis. Ideally, cross-referencing of the two types of data should be possible for all failures. In this thesis, the evaluations described in Section 4.2 and Chapter 5 were performed using processor halt logs. All of the other evaluations were performed using human-generated software failure reports.

Human-generated software failure reports used in this study were extracted from the Tandem Product Report (TPR) database, a component of the Tandem Product Reporting System (PRS) [32]. A TPR is used to report all problems, questions, and requests for enhancements by customers or Tandem employees concerning any Tandem products. Figure 2.2 shows a sample TPR. A TPR consists of a header and a body. The header provides fixed fields for the information, such as the date, problem type, urgency, customer and system identifications, and brief problem description. The body of a TPR is a textual description of all actions taken by Tandem analysts in diagnosing the problem. If a TPR reports a software failure, the body also includes the log of memory dump analyses performed by Tandem analysts. A software failure can occur during the testing and operational phases. In this research, only the software failures that occurred during the operational phase (i.e., reported from user systems) were used.



Tandem Product Report

TPR number:: 91-01-03 17:50 Seveirity: 2

Product Name: GUARDIAN Kernel
Classification: software problem

Origination: ABC Financial Inc.
777 Lawrence Street
Chicago, IL 60661

Date Received: 91-01-03 14:54
Date Returned: 91-01-10 10:49

System Number: 0056983

Accompanying Information: dump file location \ABC.prs.jan031750.*

Problem Description: Halts on CPUs 4 and 5.
Process $ABC runs in CPU 4 backed up in CPU 5

- -

Response:
(all actions including dump analyses taken by Tandem abnalysts to diagnose the problem)

Figure 2.2 Tandem Product Report

Human-generated software failure reports contain detailed information about the underlying faults, failure symptoms, and fixes. As a result, these reports can be used to address many software dependability issues. There are two major challenges in evaluating manual reports. First, underreporting can be significant. It is estimated that about 80% of field software failures in Tandem systems are not reported. Ideally, cross-referencing between on-line event logs generated by the operating system, manual reports, and operator logs should be possible. Second, since these reports contain textual descriptions by analysts, they cannot be readily analyzed by automatic tools. The raw data usually has to be reorganized into a structured database. This reorganization involves data categorization (i.e., generating categories and counting instances for each category), which requires understanding the details of problems long after the cases are closed, when important information may no longer be available. This challenge can be serious, because the bulk of the evaluation efforts might be spent on such data reorganization. This problem can be resolved by generating categories a priori, so that analysts can simply choose and mark the best-matching category when they close a case. Such category generation must be performed for each question.

The on-line processor halt log is a subset of the Tandem Maintenance and Diagnostic System (TMDS) event log maintained by the GUARDIAN operating system [33]. The processor halt log contains accurate records of occurrences of software failure and recovery over time. Figure 2.3 shows a sample event entry extracted from the processor halt log from a Tandem system. The information in the event was decoded to make it readable. An event record consists of a header and a body. A header contains general information, such as the time of occurrence, the subsystem and device affected, and the type of event. Typically, all errors share the same header format. In this example, the event reports a processor halt seemingly caused by a software fault. A body contains more detailed information about the event. The format of the body differs from event to event. In Figure 2.3, the body contains the apparent cause of the halt from an operating system perspective (halt error code) and a summary of the processor state at the time of halt.

Measurements were made on five Tandem systems (one field system and four in-house systems) for a total of five system years. Failures are rare in the Tandem system, and only two in-house systems had enough failures for a meaningful analysis. These systems were a Tandem Cyclone and a Tandem VLX, both used by Tandem software developers for a wide range of design and development experiments. They were operating as beta sites and were configured with old hardware. Sometimes the systems were deliberately faulted for analysis. As such, they are not representative of Tandem systems in the field.

Time	Subsystem	Device	Event
06SEP91 09:57:00	CPU	CPU-2	CPU-Software-Halt
		SECT-1	
		CAB-1	
	CPU-Type:	3	
	Halt-error-code:	%4040	
	OS-Type:	0	
	P-register:	%60544	
	E-register:	%3407	
	L-register:	%7250	
	Current-space-id:	%147	
	Coldload-address:	%351	
	Current-PCB-address:	%107200	
	PCB-base-address:	%100100	
	DDT-status:	%10,%0,%0,%343,%120	
	DDT-error-bits:	%0,%0,%0,%0,%2,% 0,%0,%0,%0,%0,%0,%210	
	Register-file:	%%4040,%0,%4317,%1 ,%0,%57,%1360,%12742,%0	
		%170000,%100,%0,%177440,%53,%20,%1	

Figure 2.3 Software Failure Event in the Processor Halt Log

On-line event logs provide close to 100% of reporting and accurate timing information on error occurrences and recovery. But there are several challenges in evaluating event logs. First, modern computer systems are reliable, and a long period of measurement (often on a number of systems) is required to conduct a meaningful analysis. The volume of data can be huge. Software tools that can automate the basic analysis steps and manage the data must be developed. Second, the meaning of a record and the format of an event in the log can sometimes differ among versions of the operating system and among machine models, which is to be expected because the software and hardware of a system evolve. Therefore, software analysis tools must be updated according to these changes. Third, the information provided by event logs may not be complete. For example, an event report of a software failure provides the apparent cause of the halt and the processor state, but the underlying fault may not be directly related to the process that experienced the problem. It is therefore necessary to supplement the information in machine-generated event logs with that in manual logs, such as software failure reports and operator logs.

Chapter 3

Fault Categorization and Characterization of Error Propagation

The first step in analyzing software failure data is to understand the underlying software faults of failures and the symptoms of software faults. Using human-generated software failure reports (i.e., a collection of memory dump analyses of field software failures), this chapter categorizes the underlying faults of software failures, identifies the immediate effects of the faults on the processor state, and traces the propagation of the effects on other system areas until problems are detected by the operating system. The issue of recurrence is briefly discussed.

3.1 Fault Categorization

A collection of faults identified on a software system naturally reflects the characteristics of the corresponding software development environment. As a result, such data can be used for fine-tuning the development environment and for improving software quality. Fault categorization is a frequently used method of addressing these issues. For this reason, results of fault categorizations can be regarded as measurement-based software fault models. Many studies have performed fault categorization using faults identified during the development phase [4], [3], [6]. The motivations for such categorizations are shared in studies of both development phase and operational phase data. However, the software fault profile of operational software can be quite different from that of software during the development phase, due to differences in the operational environment and software maturity. Therefore, it is important to investigate the fault profile of operational software.

We studied the underlying causes of 200 TPRs that reported processor failures seemingly due to faults in the Tandem system software [34]. These TPRs include all software failures reported by users during a time period in 1991. Table 3.1 shows a breakdown of the 200 TPRs based on cause types. The numbers inside parentheses further subdivide problem types. Twenty-one of the 200 TPRs were generated due to nonsoftware causes. The underlying causes of these failures indicated that hardware and operational faults sometimes cause failures that appear to be caused by software faults. Our experience shows that determining whether a failure is caused by software faults is not always straightforward, which is partly because of the complexity of systems and partly because of close interactions between the software and hardware of the system. In 26 of the remaining 179 TPRs, analysts believed that the underlying problems were software faults, but they had not yet located the faults. These cases are referred to as *unidentified* problems.

Table 3.1 Problem Types

Problem Type	#TPRs	
Software problem		179
– Cause (software fault) identified	(153)	
– Cause unidentified	(26)	
Nonsoftware problem		21
– Operational fault	(10)	
– Hardware fault	(8)	
– Resource shortage due to high activity	(2)	
– Documentation (Manual) fault	(1)	
All		200

Table 3.2 shows the results of a fault classification using the 153 TPRs whose software causes were identified. The table shows both the number of TPRs and the number of unique faults. The differences between the two represent multiple failures caused by the same fault. Table 3.2 shows what kinds of faults the developers introduced. In the

table, the faults were ordered by the difficulty of testing and identifying them. "Incorrect computation" refers to an arithmetic overflow or the use of an incorrect arithmetic function (e.g., use of a signed arithmetic function instead of an unsigned one). "Data fault" refers to the use of an incorrect constant or variable. "Data definition fault" refers to a fault in declaring data or in defining a data structure. "Missing operation" refers to an omission of a few lines of source code. "Side effect of code update" occurs when not all dependencies between software modules were considered when updating software. "Unexpected situation" refers to cases in which software designers did not anticipate a legitimate operational scenario, and the software did not handle the situation correctly. Table 3.2 shows that "Missing operation" and "Unexpected situation" are the most common types of software faults in Tandem systems. Additional code inspection and testing efforts can be directed for identifying such faults.

Table 3.2 Software Fault Categorization

Fault Category	#Faults	#TPRs
Incorrect computation	3	3
Data fault	12	21
Data definition fault	3	7
Missing operation:	20	27
– Uninitialized pointer	(6)	(7)
– Uninitialized nonpointer variable	(4)	(6)
– Not updating data structure on the occurrence of event	(6)	(9)
– Not telling other processes about the occurrence of event	(4)	(5)
Side effect of code update	4	5
Unexpected situation:	29	46
– Race/timing problem	(14)	(18)
– Errors with no defined error-handling procedures	(4)	(8)
– Incorrect parameter or invalid call from user process	(3)	(7)
– Not providing routines to handle legitimate but rare operational scenarios	(8)	(13)
Microcode defect	4	8
Other (cause does not fit any of the above class)	10	12
Unable to classify due to insufficient information	15	24
All	100	153

A high proportion of simple faults, such as incorrect computations or missing operations, is usually observed in new software, while a high proportion of complex faults, such as unexpected situations, is usually observed in mature software. The coexistence of a significant number of simple and complex faults is not surprising, because the measured system is a large software system consisting of both new and mature components. Further, some user systems run earlier versions of software, while other user systems run later versions. Nontheless, one would like to see fewer simple faults. The existence of a significant proportion of simple faults indicates that there is room for improvement in the code inspection and testing process.

A software failure caused by a newly found fault is referred to as a *first occurrence*; a software failure caused by a previously reported fault is referred to as a *recurrence*. The 153 TPRs whose software causes were identified occurred due to 100 unique faults (Table 3.2). Out of the 100 unique faults, 57 were diagnosed before our measurement period. Therefore, 43 new software faults were identified during the measurement period. That is, about 72% (110 out of 153) of the TPRs reported recurrences of previously reported software faults. When one considers that a single TPR may list a rapid succession of failures, which are likely to be caused by the same fault, the actual percentage of recurrences may be higher. This result shows that the number of faults in software (which can be regarded as the failure rate when only a single copy of the software runs) is not the only important factor. The dependability of operational software can be significantly improved by reducing the number of recurrences or by efficiently handling recurrences. This issue is discussed further in Chapters 6 and 7.

3.2 Software Error Propagation

Given that complete elimination of software faults in a large, continually evolving software system is difficult, it is important that the software handles the effects of software faults efficiently. Such a design requires understanding the effects of software faults and establishing efficient software fault models. While efficient models for hardware faults exist, the issue of software fault models is open.

Software fault models can be built from two perspectives: software engineering and software fault tolerance. Examples of software fault models built from the software engineering perspective are the results of software fault categorization. Such models can be used for fine-tuning the software development environment and for avoiding and eliminating software faults. Software fault models built from the software fault tolerance perspective are considered essential for designing efficient error detection, diagnosis, and recovery strategies. These models can be built based on a knowledge of faults, the effects of software faults (i.e., errors), propagation characteristics of error, or a combination of these factors.

With the above in mind, we reverse-engineer field software failures to recreate the error propagation process (Figure 2.1) in this section. We identify the immediate effects of software faults on the processor state and trace the propagation of the effects to other system areas, until problems are detected by the operating system. The error propagation modes developed in this section will be used for analyzing the symptoms of recurrences, for the purpose of developing, in Chapter 7, a symptom-based strategy for automatically diagnosing recurrences.

3.2.1 First errors

The term *first error* is defined as the immediate effect of a software fault on the processor state when the fault is exercised. In other words, the first error of a software fault refers to the first program variable that acquires an incorrect value because of the fault. We identified the first errors from the 153 TPRs whose software causes were diagnosed and classified them into the five categories:

(1) Single address error: an incorrect address word is developed.

(2) Single nonaddress error: an incorrect nonaddress value is developed; instances in this category were further divided into four subclasses: incorrect field size, incorrect index, incorrect flag, and the rest (other).

(3) Multiple errors: multiple errors are generated at once; instances in this category were further divided into two subclasses: random corruption in a memory area without regard to the data structure (e.g., a corruption caused by a stack area overlap or a missing initialization of a memory area), and multiple regular errors in data structure (e.g., memory management tables become inconsistent due to a partial update or a request buffer is overwritten by another request).

(4) Other: the first error does not fit any of the above categories (e.g., an invalid request caused by a race condition).

(5) Unable to classify: the first error could not be identified due to insufficient information in the TPRs.

Table 3.3 shows the results of the classification. Most single address errors are related to pointers; typically, random (unpredictable) addresses are generated. An incorrect field size can subsequently cause corruption of a memory area, and an incorrect table or array index can subsequently develop an incorrect address. Invalid requests caused by race conditions or illegal procedure calls made by user processes were the major instances in the group "Other."

3.2.2 Propagation modes

Using the information in TPRs, we attempted to reconstruct parts of the error propagation process (Figure 2.1). Complete reconstruction of the error propagation process for a failure is not possible even with a memory dump. We focused on two aspects of the failures: error latency and the propagation characteristics of first errors. Because of the complexity of the error propagation process and the nature of the data, it was not possible to quantify the error latency with a high degree of precision. But it was

Table 3.3 First Errors

Category	Fraction (%)
Single address error	23
Single nonaddress error	38
– Field size	(8)
– Index	(10)
– Flag	(6)
– Other	(13)
Multiple errors	18
– Random corruption	(7)
– Data structure error	(11)
Other	9
Unable to classify	12

possible to determine whether a problem was detected before the task (i.e., service to a request) causing the first error was completed (short error latency versus significant error latency).

It was also possible to classify the propagation characteristics of first errors into three groups: *no propagation*, *further corruption*, and *quick detection.* No propagation refers to cases in which there is no possibility of error propagation, i.e., the first error is certain to be detected on the first access. Figure 3.1 shows a real example of no propagation. In the figure, a dotted line represents a latency. The SIOP software implements a data communication protocol. There was a program path in the SIOP software wherein the deletion of an element of a linked list was requested twice. When the path was exercised, the system software detected the problem and asserted a processor halt on the second request, because the element did not exist in the linked list.

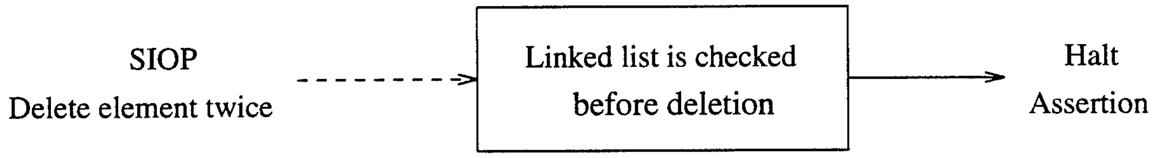


Figure 3.1 No Propagation

Further corruption refers to error propagation across processes and the generation of more errors. Figure 3.2 shows an example of further corruption. In the figure, process PK is an execution of a tool to check and change a processor configuration, such as the number of active processes and I/O lines. Process MS collects resource usage data, and process TM is in charge of concurrency control and failure recovery. When the operator ran PK with a certain option, which is not frequently used, PK used an incorrect constant to initialize its data structure. As a result, PK overwrote (cleared) the page addresses of the first segment in the segment page table. The first segment is owned by MS, and MS was running on the processor. When MS stored resource usage data, it used incorrect addresses (addresses of zero) and corrupted the system global data. A processor halt occurred due to address violation when TM accessed and used the address of a system data table.

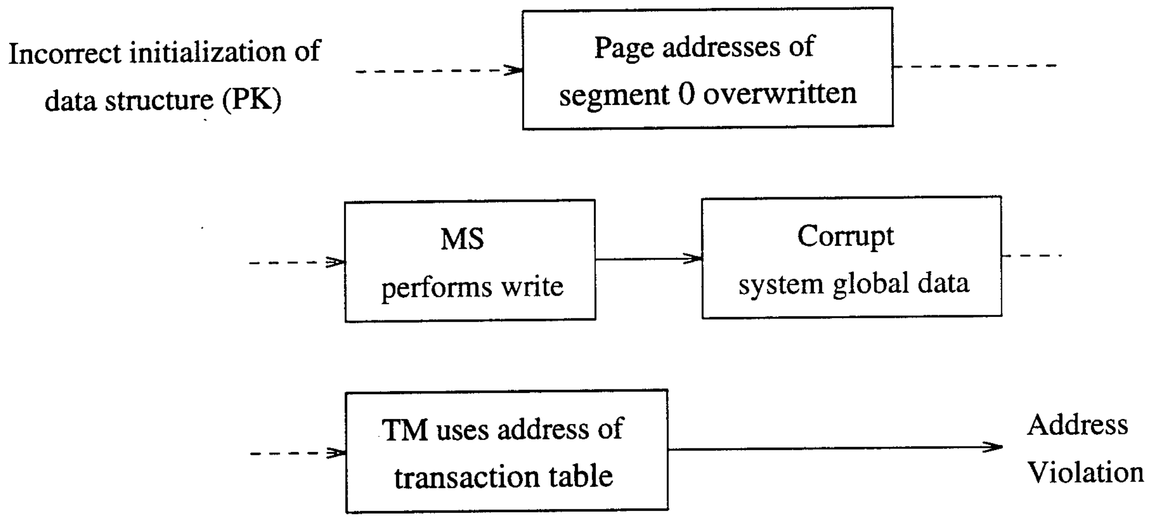

Incorrect initialization of data structure (PK)
Page addresses of segment 0 overwritten
MS performs write
Corrupt system global data
TM uses address of transaction table
Address Violation

Figure 3.2 Further Corruption

Quick detection lies between the above two propagation modes. In this situation, there is no guarantee that there will be no propagation. The problem is detected quickly, after the first error is accessed for the first time, while the task that made the first access is executed. Figure 3.3 shows an example of quick detection. The XIOP software implements a data communication protocol. There was an uninitialized pointer in a parity error-handling routine of the XIOP software. When a parity error occurred during a data transfer, the uninitialized pointer was used, and a processor halt occurred after a few program statements due to an illegal address reference.

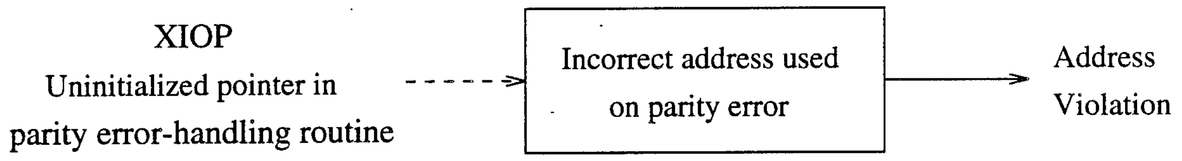


Figure 3.3 Quick Detection

Table 3.4 shows the results of a classification of the 153 TPRs based on error propagation modes and error latency. A potential danger of significant error latency is that if a problem is detected long after the first error is developed, identification of the underlying software fault can be difficult. A similar difficulty exists in the further corruption cases. Once many errors are generated, identification of the underlying software fault can be difficult. The propagation mode is typically determined by the code characteristics and the processor state when a first error is generated and accessed. Table 3.4 shows that consistency checks made by the operating system significantly change the propagation mode profile by changing the code characteristics. The error propagation modes developed here will be used for analyzing the symptoms of recurrences in Subsection 7.1.2.

Table 3.4 Propagation Modes

Category	Fraction (%)
Short error latency	47
– No propagation	(18)
– Quick detection	(29)
Significant error latency	41
– No propagation	(13)
– Quick detection	(10)
– Further corruption	(18)
Other	1
Unable to classify	11

3.2.3 Problem detection

The 153 TPRs were also classified into four groups, based on how the operating system detected the problems (Table 3.5). An address violation occurs when the operating system detects an illegal address reference by a privileged process. This detection mechanism is common in most computer systems. An address violation can be detected during a page fault interrupt service or during an instruction execution in the form of an instruction failure. In the table, the term nonkernel represents all Tandem software products running as privileged processes and forming the outer layer of the GUARDIAN operating system. Table 3.5 shows that, in 52% of software failures, problems are detected by consistency checks made by the operating system.

Table 3.5 Problem Detection

Detection Mechanism	Fraction (%)
Address violation	48
Kernel consistency check	19
Nonkernel consistency check	33
Other	1

3.2.4 Error propagation model

Figure 3.4 provides an overall picture of error propagation, from underlying software faults to problem detection, by relating the information in Tables 3.2 through 3.5. A circle or a rectangle represents a category, and the numbers inside it represent the number of TPRs in that category and its percentage of the 153 TPRs. An arrow represents a transition, and the associated number represents a branching probability from the source state. For example, data faults account for 14% of the faults, and if a fault in this category is exercised, there is a 24% chance that an incorrect address will be generated. Figure 3.4 captures all major error propagation paths that must be eliminated.

In Figure 3.4, "Unexpected Situation" and "Single Nonaddress" error were not split further, because there were not enough instances in each subcategory and further

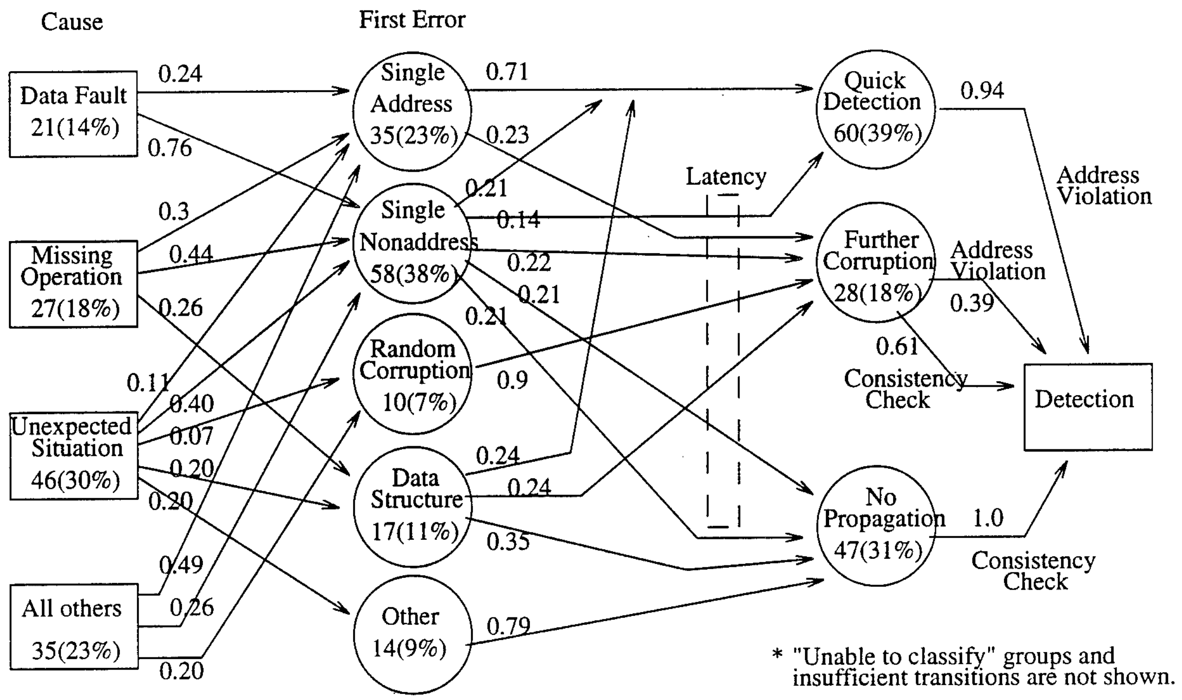


Figure 3.4 Error Propagation Model

divisions did not reveal any patterns. The dotted box in the figure represents a significant error latency, and an arrow that crosses the dotted box represents a transition with a significant latency. Circles or rectangles representing "Unable to classify" groups and arrows representing insignificant transitions are not shown in the figure.

All instances of uninitialized pointer in "Missing Operation" group naturally generate single address errors. About 40% of the instances of not updating data structure on the occurrence of event or not telling other processes about the occurrence of event in "Missing Operation" group generate multiple regular errors in data structures. These instances account for the transition from "Missing Operation" to "Data Structure" error. Instances of "Unexpected Situation" generate a variety of first errors. Among these, invalid requests caused by race conditions or illegal procedure calls made by user processes account for the transition to "Other" first error group. Figure 3.4 shows that address errors are difficult to handle with consistency checks. The data showed no instances in which "Single Address" error is guaranteed to be detected on the first access.

In Figure 3.4, “No Propagation” is a desired state, because it does not threaten the integrity of the data in the system. It is a significant state in the measured system because of the use of redundant data structures and consistency checks. All instances of “No Propagation” are detected by consistency checks. In 94% of the instances of “Quick Detection,” problems are detected due to address violations; the rest are detected by consistency checks.

“Further Corruption” is a dangerous state, in that error propagation can occur recursively and multiple errors are generated until the problem is detected. Some of the errors may break the fault containment boundary assumed for on-line recovery and thus cause another problem later. Any process that accesses corrupted data can potentially assert a halt, and thus a single fault can cause a variety of failure symptoms, which may complicate the diagnosis.

3.2.5 Symptoms of unidentified failures

Unidentified failures account for about 15% of software failures (Table 3.6). Unidentified failures can cause congestion and stress to software service. The *recreate* problem in identifying and diagnosing software failures in the field was discussed in [21]. Table 3.6 summarizes the symptoms of the unidentified failures. In the majority of these failures, one or more memory locations were overwritten with invalid data for unknown reasons. In seven failures, problems were detected due to inconsistencies in data structures. A data structure inconsistency indicates that sections of a data structure contain conflicting information. A data structure inconsistency is probably caused by a partial update of data structure. Enforcing a stronger data encapsulation rule can reduce the number of this type of failures. In three cases, some analysts suspected transient hardware faults.

3.3 Summary

This chapter explores new ground for building software fault models from the software fault tolerance perspective. In addition to categorizing the underlying faults of software

Table 3.6 Symptoms of Unidentified Failures

Symptom	# TPRs
Data structure inconsistency	7
Data overwritten	17
- One word	(8)
- Multiple words	(9)
Other	2

failures, we identified the immediate effects of the faults on the processor state (i.e., first errors) and traced the propagation of the effects on other system areas until problems were detected by the operating system, using a collection of memory dump analyses of field software failures.

The results showed that about 72% of reported field software failures in Tandem systems are recurrences of previously reported faults. This shows that, in environments where many users run the same software, the number of faults in software is not the only important factor. Recurrences can seriously degrade software dependability in the field. Clearly, the impact of recurrences on system dependability must be modeled and evaluated. Missing operations and not providing routines to handle rare but legitimate operational scenarios are the most common types of software faults in Tandem systems. Additional code inspection and testing efforts can be directed for identifying these faults. The results showed the coexistence of a significant proportion of simple and complex faults, which is not surprising because the measured system is a large software system consisting of both new and mature components. The existence of a significant proportion of simple faults indicates that there is room for improvement in the code inspection and testing process.

The data showed that there is a 60% chance that a single program variable acquires an initial, incorrect value when software faults are exercised. In about 20% of the cases, multiple program variables are affected simultaneously. Once errors are generated, the three major error propagation modes are: the first error is certain to be detected on the first access by consistency checks (no propagation, 31%); the problem is detected shortly after the first error is accessed and used (quick detection, 39%); and the first error causes more errors, which are detected after a significant latency (further corruption, 18%). In about half of the failures, problems are detected by consistency checks; in the other half, problems are detected due to address violations. The error propagation modes developed will be used for analyzing the symptoms of recurrences in order to develop (in Subsection 7.1.2) a symptom-based strategy for automatically diagnosing recurrences.

Chapter 4

Evaluation of Software Fault Tolerance

This chapter evaluates the software fault tolerance of process pairs in the Tandem GUARDIAN system. Two types of analyses are performed. First, the level of software fault tolerance achieved by the use of process pairs and the detailed reasons for software fault tolerance are investigated using human-generated software failure reports. Next, the impact of software failures on system performance and the effectiveness of the built-in single-failure tolerance of the Tandem system against software failures are evaluated by conducting Markov reward analysis using on-line processor halt logs.

4.1 Software Fault Tolerance of Process Pairs

It has been observed that process pairs allow the Tandem system to tolerate certain software faults [27], [1]; that is, in many processor halts caused by software faults, the backup of a failed primary can continue the execution. This observation is rather counter-intuitive, because the primary and backup run the same copy of the software. The phenomenon was explained by the existence of subtle faults, often referred to as *transient* software faults, that are not exercised again on a restart of the failed software. Field software faults were not identified during the testing phase, and many of them could be transient in nature. Since the technique is not explicitly intended for tolerating software faults, study of field data is essential for understanding the phenomenon and for measuring the effectiveness of the technique for tolerating software faults.

This section uses human-generated field software failure reports to investigate the user-perceived ability of the Tandem system to tolerate faults in its system software [34]. Two conditions must be mentioned. First, only the faults in the system software that cause processor failures are considered. Thus, we are not looking at the entire set of software faults. Second, only the software fault tolerance achieved specifically by the use of process pairs is considered.

The evaluation is important because, although process pairs are specific to Tandem systems, it is an implementation of the idea of checkpointing and restart, which is a general approach. Clearly, software dependability can be improved by designs exploiting such knowledge in similar environments. This evaluation also important because there are no efficient techniques available for achieving software fault tolerance in large, continually evolving software systems. Recently, attempts have been made to make use of the transient nature of some software faults for tolerating these faults in user applications using checkpointing and restart [35], [36].

4.1.1 Measure of software fault tolerance

There were 179 TPRs generated because of software faults during the measured period (Section 3.1). Since each TPR reports just one problem, sometimes two TPRs are generated as a result of a multiple processor halt. There were five of these cases, making a total of 174 software failures during the measured period. Table 4.1 shows the severity of the 174 software failures. A single processor halt implies that the built-in single-failure tolerance of the system masked the software fault that caused the halt. All multiple processor halts were grouped because, in the Tandem system, a double processor halt can potentially cause additional processor halts because of the system architecture. That is, if the system loses a set of disks as a result of a double processor halt and the set of disks contains files required by other processors, additional halts can occur in the other processors. (In the Tandem system, a disk is connected with two processors through dual-port disk controllers.) There was one case in which a software failure occurred in the middle of a system coldload.

Table 4.1 Severity of Software Failures

Severity	# Failures
Single processor halt	138
Multiple processor halt	31
During system coldload	1
Unable to classify	4
All	174

In this evaluation, the term software fault tolerance (SFT) refers to the system's ability to tolerate software faults. Quantitatively, it is defined as

$$SFT = \frac{\text{number of software failures in which a single processor is halted}}{\text{total number of software failures}} . \quad (4.1)$$

SFT represents the user-perceived ability of the system to tolerate faults in its system software due to the use of process pairs.

Table 4.1 shows that process pairs provide a significant level of software fault tolerance in distributed transaction-processing environments. The measure of software fault tolerance is estimated to be 82% (138 out of 169). This measure is based on reported software failures. The issue of underreporting was discussed in [1]. The consensus among experienced Tandem engineers is that about 80% of software failures are not reported as TPRs and that most of them are single processor halts. If that assessment is true, then the software fault tolerance may be as high as 96%.

4.1.2 Outages due to software

This evaluation first focused on the multiple processor halts. For each multiple processor halt, we investigated the first two processor halts to determine whether the second halt occurred on the processor executing the backup of the failed primary process. In these cases, we also investigated whether the two processors halted because of the same software fault.

Table 4.2 shows that in 86% (24 out of 28, excluding "Unable to classify" cases) of the multiple processor halts, the backup of the failed primary process was unable to continue the execution. In 81% (17 out of 21, excluding "Unable to classify" cases) of these halts, the backup failed because of the same fault that caused the failure of the primary. In the remaining 19% of the halts, the processor executing the backup of the failed primary halted because of another fault during job takeover. The level of software fault tolerance achieved with process pairs is high, but not perfect. A single fault in the system software can manifest itself as a multiple processor halt, which the system is not designed to tolerate. About half of the multiple processor halts resulted in system coldloads. The data showed that, in most situations, the system lost a set of disks that contained files required by other processors as a result of the first two processor halts, and other processors also halted. This result shows the major failure mode of the system because of software.

Table 4.2 Reasons for Multiple Processor Halts

Reasons for Multiple Processor Halts	# Failures
The second halt occurs on the processor executing the backup of the failed primary.	24
– The second halt occurs due to the same fault that halted the primary.	(17)
– The second halt occurs due to another fault during job takeover.	(4)
– Unable to classify.	(3)
The second halt is not related to process pairs.	4
– The system hangs.	(1)
– Faulty parallel software executes.	(1)
– There is a random coincidence of two independent faults.	(1)
– A single processor halt occurs, but system coldload is necessary for recovery.	(1)
Unable to classify.	3
All	31

4.1.3 Characterization of software fault tolerance

The information in Table 4.1 poses the question of why the Tandem system loses only one processor in 82% of software failures and, as a result, tolerates the software faults that cause these failures. We identified the reasons for software fault tolerance in all single processor halts and classified them into several groups. Table 4.3 shows that, in 29% of single processor halts, the fault that causes a failure of a primary process is not exercised again when the backup reexecutes the same task after a takeover. These situations occur because some software faults are exposed in a specific memory state (e.g., running out of buffer), on the occurrence of a single event or a sequence of asynchronous events during a vulnerable time window (timing), by race conditions or concurrent operations among multiple processes, or on the occurrence of a hardware error.

Table 4.3 Reasons for Software Fault Tolerance

Reasons for Software Fault Tolerance	Fraction (%)
The backup reexecutes the failed task after takeover, but the fault that caused a failure of the primary is not exercised by the backup.	29
– Memory state	(4)
– Timing	(7)
– Race or concurrency	(6)
– Hardware error	(4)
– Others	(7)
The backup, after takeover, does not automatically reexecute the failed task.	20
It is the effect of error latency.	5
A fault stops a processor running a backup.	16
The cause of a problem is unidentified.	19
Unable to classify.	12

Figure 4.1 shows a real example of a fault that is exercised in a specific memory state. The primary of an I/O process pair, which is represented by SIOP(P) in the figure, requested a buffer to serve a user request. Because of the high activity in the processor executing the primary, the buffer was not available. However, because of a

software fault, the buffer management routine returned a "successful" flag, instead of an "unsuccessful" flag. The primary used the returned, uninitialized buffer pointer, and a halt occurred in the processor running the primary because of an illegal address reference by a privileged process. Clearly, such a situation was not tested during the development phase. Since a memory dump is taken only from a halted processor in a production system, a memory dump of the processor running the backup is not available. Our best guess is that the backup process served the request again after takeover but did not have a problem, because a buffer was available on the processor running the backup.

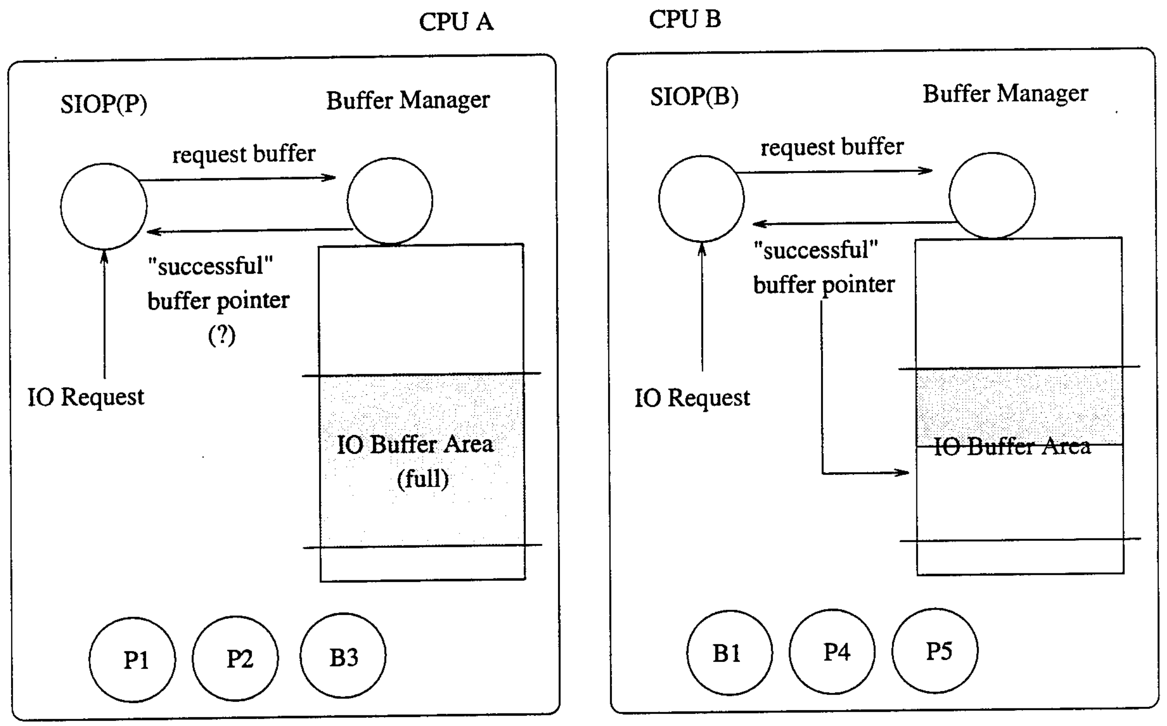


Figure 4.1 Differences between the Primary and Backup Executions

Table 4.3 also shows that, in 20% of single processor halts, the backup of a failed primary process does not serve the failed request after a successful takeover, because some faults are exposed while serving requests that are important but are not automatically resubmitted to the backup upon a failure of the primary. Figure 4.2 illustrates an example of these situations. (This example was discussed in Subsection 3.2.2, Figure 3.2.) In the figure, process PK is an execution of a tool to check and change a processor configuration, such as the number of active processes and I/O lines. Process PK does not run as a process pair because, if the processor being monitored or reconfigured halts while executing PK, there is no need to monitor or reconfigure the halted processor any longer. Process MS collects resource usage data, and process TM is in charge of concurrency control and failure recovery. Both MS and TM run as process pairs.

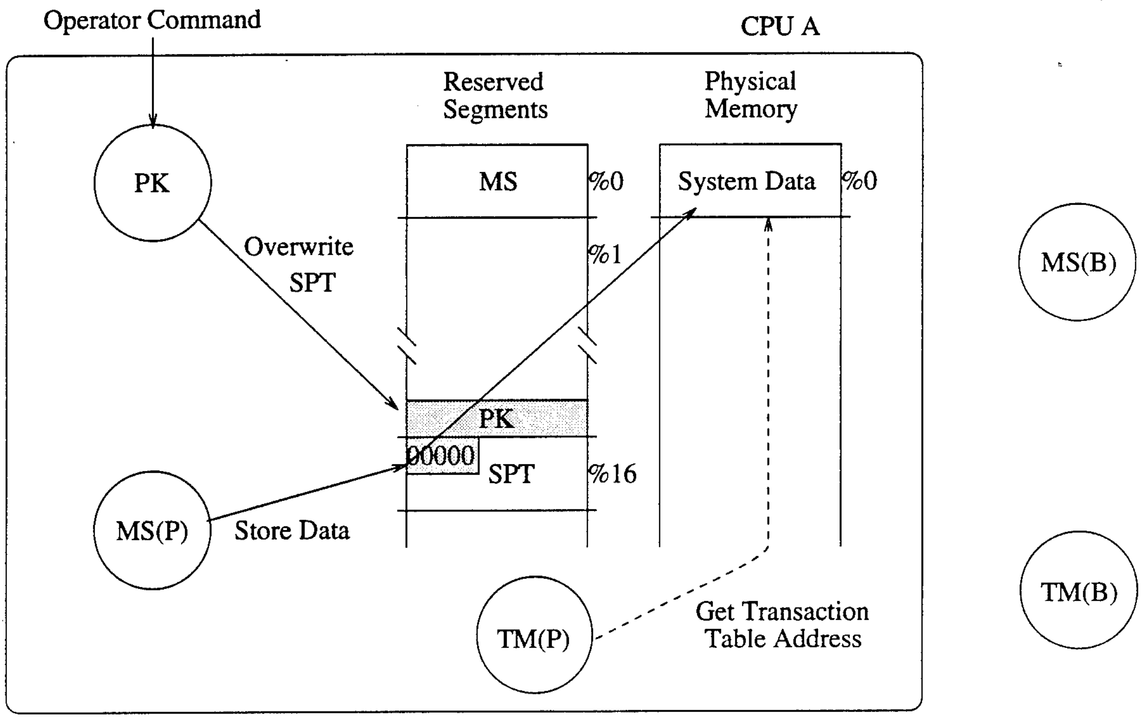


Figure 4.2 Faults Exposed by Non-Process Pairs

When the operator ran PK with a certain option, which is not frequently used, PK used an incorrect constant to initialize its data structure. As a result, it overwrote (cleared) the page addresses of the first segment in the segment page table. The first segment is owned by MS, and MS was running on the processor. When MS stored resource usage data, it used incorrect addresses (addresses of zero) and corrupted the system global data. A processor halt occurred as a result of an address violation when TM accessed and used the address of a system data table. The backups of failed primaries would take over, but they would not have problems, because PK was running only on the halted processor.

Another example of faults exposed by tasks that do not run as process pairs is the faults that cause processor failures during the execution of the operator requests for reconfiguring I/O lines. Utilities to perform these reconfigurations run as process pairs, but the operator command such as "add a line" is not automatically resubmitted to the backup, because it is an interactive task that can easily be resubmitted by the operator if the primary fails. Suppose that an operator's request to add an I/O line caused a failure of the primary. In this situation, the operator would typically recover the halted processor, rather than submit the same request to the backup. If the operator wants to repeat the same request, he or she would normally repeat it on the primary after the halted processor is reloaded. If the operator submits the request to the backup instantly upon a failure of the primary, one of two situations can be expected: the backup also halts, or the backup serves the request without any problem because of the reasons in Table 4.3.

In the above examples, the task (i.e., process PK or a command to add an I/O line) does not survive the failure. But process pairs allow the other applications on the halted processor to continue to run. This situation is not strictly software fault tolerance but a side benefit of using process pairs. If these failures are excluded, the estimated measure of software fault tolerance is reduced to 77%.

Another reason for the software fault tolerance is that some software faults cause errors that are detected after the service that caused the errors finishes successfully (effect of error latency). Figure 4.3 shows an example of these situations. The figure shows a data transfer between two primary I/O processes: SIOP(P) and XIOP(P). The underlying software fault is an extra line in the SIOP software that causes SIOP(P) to transfer one more byte than is necessary. This fault does not always cause a problem, because the size of a buffer is usually bigger than the size of a message. When a message and a buffer had equal sizes, the first byte in the end tag of the buffer was overwritten. This corruption did not affect the data transfer, because tags are not a part of data area. (The tags are used to check the integrity of a data structure, but for performance reasons, they are not checked after every data transfer.) The data transfer was completed and checkpointed to the backup. The corruption in the end tag was found later, when SIOP(P) returned the buffer to the buffer manager. The buffer manager checked the integrity of the begin and end tags, found a corruption, and asserted a halt of the processor it runs on ("CPU A" in the Figure 4.3). The backups of the failed primaries would take over, but they would not have problems because the data transfer that caused the error was already completed. The difference between this case and the first group of cases listed in Table 4.3 is that the software function that caused the failure of the primary did not have to be executed again in the backup.

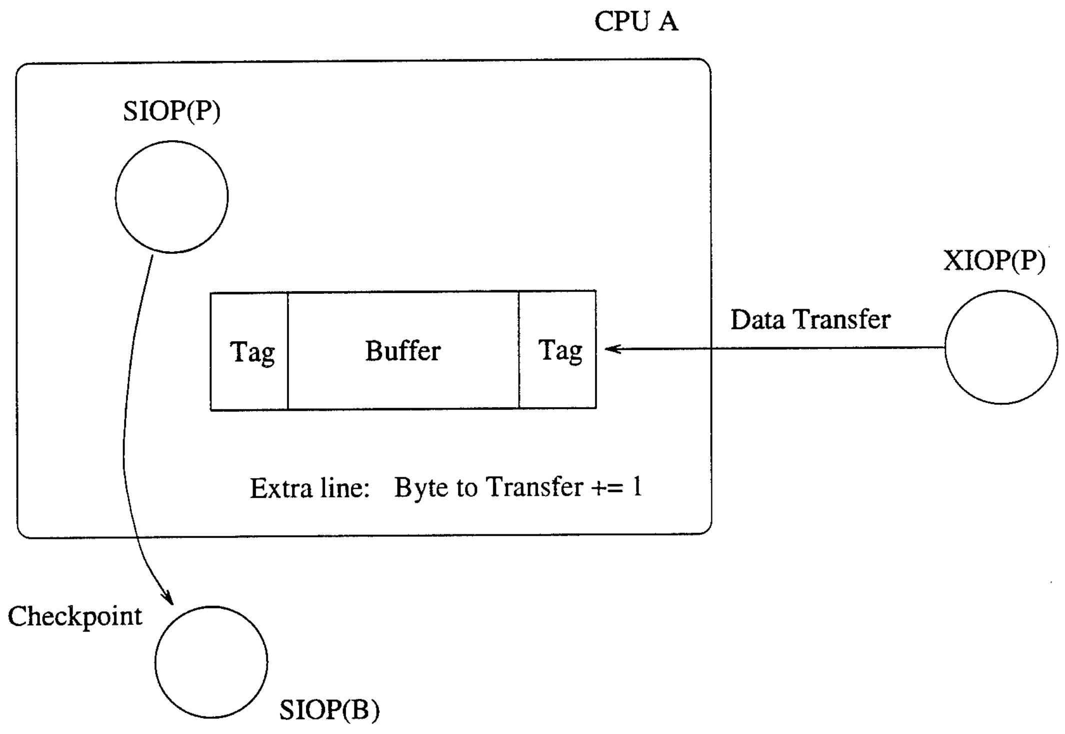


Figure 4.3 Effect of Error Latency

Table 4.3 also shows that 19% of single processor halts are failures of backup processes. This result indicates that the software fault tolerance does not come without cost; the added complexity due to the implementation of process pairs introduces additional software faults into the system software. The estimated measure of software fault tolerance (77%) is adjusted again to 72% when these failures are excluded. All unidentified failures were single processor halts, which is understandable, because these are caused by subtle faults that are very difficult to observe and diagnose. The reason that an unidentified problem caused a single processor halt is unknown. Based on their symptoms, we speculate that a significant number of unidentified problems were single processor halts because of the effect of error latency.

4.1.4 Discussion

The results in this section have several implications. First, subtle faults exist in all software, but software fault tolerance is not achieved if the backup execution is a replication of the original execution (the same sequence of events in the same processor state). The results show that the source of software fault tolerance lies in differences between the original and backup executions in the processing environment (i.e., the processor state and the sequence of events). The loose coupling between processors in the measured system provides these differences. This result confirms that there is another dimension for achieving software fault tolerance in distributed environments. The actual level of software fault tolerance achieved by the use of process pairs will depend on the degree of difference between the original and backup executions in the processing environment. Each processor in a Tandem system has an independent processing environment; therefore, the system naturally provides such differences. (The advantages of using checkpointing, as compared with lock-step operation, in tolerating software faults was discussed in [27].) The high level of software fault tolerance observed is probably also because the measured operating system is a mature software system that has primarily subtle faults.

Second, the results indicate that process pairs can also allow the system to tolerate nontransient software faults, because software failures can occur while the system executes important tasks that are not automatically resubmitted to the backup on a failure of the primary. In this case, the failed task does not survive, but process pairs allow the other applications on the failed processor to survive.

Third, short error latency with error confinement within a transaction is desirable [37]. In actual designs, such a strict error confinement might be rather difficult to achieve.

In Tandem systems, the unit of error confinement is a processor, not a transaction [27]. Errors generated during the execution of a transaction may be detected during the execution of another transaction. Interestingly, long error latency and error propagation across transactions sometimes help the system tolerate software faults. This result should not be interpreted to suggest that long error latency or error propagation across transactions is a desirable characteristic, but rather as a side effect of the system having subtle software faults. Long error latency and error propagation across transactions can make both on-line recovery and off-line diagnosis difficult.

Finally, an interesting question is: if process pairs are good, are process triples better? Our results show that process triples may not necessarily be better, because the faults that cause double processor halts with process pairs may cause triple processor halts with process triples.

4.1.5 First occurrences vs. recurrences

Table 4.4 compares the severity of the three types of software failures using the 174 software failures. There were two special cases ("Others") in the table: a multiple processor halt that occurred because of a parallel execution of faulty code (a system coldload was not required), and a software failure that occurred in the middle of a system coldload. With only a single observation in each case, the significance of these situations is unclear, and they were not considered in the subsequent analysis.

Table 4.4 Severity of Software Failures by Failure Type

Failure Type	#Failure Instances	#Double CPU Halts	#System Coldloads	Severity Unclear	#Others
First occurrence	41	9	6	1	1
Recurrence	107	19	12	3	1
Unidentified	26	0	0	0	0

Table 4.4 indicates that a recurrence is slightly less probable to cause a double processor halt than is a first occurrence. The binomial test was used to test this observation, because it does not require an assumption about the underlying distribution to construct a confidence interval [38]. Each failure was treated as a random trial with the probability of a double processor halt being 0.23 (9 out of 39, following the statistics for the first occurrence). The hypothesis that the probability of a recurrence causing a double processor halt is equal to that of a first occurrence causing a double processor halt was tested, by calculating the probability of having 19 or fewer double processor halts out of 103 trials. The p-value was 0.16; that is, the hypothesis was rejected at the 20% significance level. Although the trend is not strong, a recurrence is less probable to cause a double processor halt than is a first occurrence. This result is probably because, if a fault is likely to cause a double processor halt (a possible outage in Tandem systems), it gets more attention and a fix is propagated and installed more promptly. The result is that the fault is less probable to cause recurrences over the long run. Table 4.4 also indicates that a recurrence is less probable to cause a system coldload than is a first occurrence. The p-value was 0.18 in this case.

Two of the six system coldloads due to first occurrences were single processor halt situations (Table 4.2). These two failures capture the secondary failure mode of the system because of software, wherein a system is coldloaded to recover from a severe, single processor halt.

4.2 Impact of Software Failures on Performance

One key measure in evaluating gracefully degraded systems is the impact of failures on system performance or service capacity. Performability models [39] and reward models [40] have been widely used to evaluate performance-related dependability measures in recent years. To evaluate the loss of service incurred by software failures and the effectiveness of built-in single-failure tolerance of the Tandem system against software failures, a Markov reward analysis was performed [41], [42]. The processor halt log collected from a 16-processor, in-house Tandem Cyclone system was used (Chapter 2). The measurement period was 23 months.

We built a continuous-time Markov model using the processor halt log. Figure 4.4 shows the model structure. In the figure, S_i represents the system state in which there are i failed processors because of software faults, and n represents the total number of processors in the system. In the model, transition rates between states were estimated using

$$r_{ij} = \frac{\text{total number of transitions from } S_i \text{ to } S_j}{\text{cumulative time the system was in } S_i}. \tag{4.2}$$

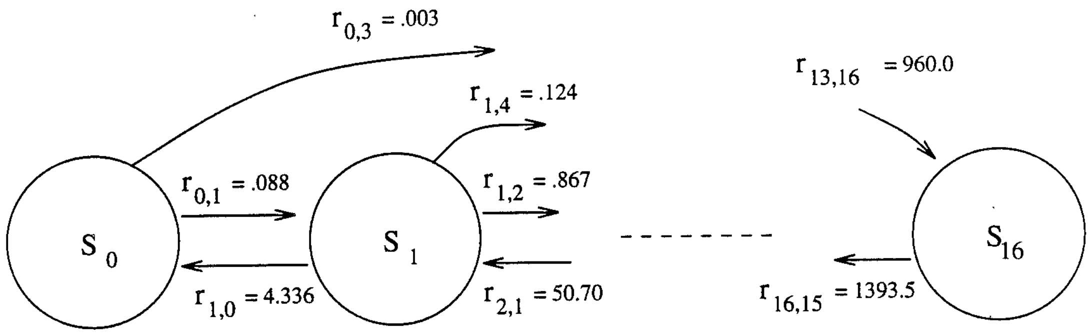


Figure 4.4 Measurement-Based Markov Model

Two reward functions were defined in the analysis. The first function (NSFT) assumes no fault tolerance. In this function, each processor halt causes degradation, and the loss of service is proportional to the total number of processors halted. The second function (SFT) reflects the fault tolerance of the Tandem system. In this function, the first processor halt causes no degradation. For additional processor halts, the loss of service is proportional to the number of processors halted. The difference between the two functions allows an evaluation of the improvement in service achieved by providing the built-in single-failure tolerance.

NSFT (No Single-Failure Tolerance):

$$r_i = 1 - \frac{i}{n} \qquad \text{if } 0 \le i \le n \tag{4.3}$$

SFT (Single-Failure Tolerance):

$$r_i = \begin{cases} 1 & \text{if } i = 0 \\ 1 - \frac{i-1}{n} & \text{if } 0 < i < n \\ 0 & \text{if } i = n \end{cases} \tag{4.4}$$

Given the Markov reward model described above, the expected steady-state reward rate, Y, can be estimated from [40]

$$Y = \sum_i p_i r_i \,, \tag{4.5}$$

where p_i is the steady-state probability of the system being in state i. The steady-state reward rate represents the relative amount of useful service the system can provide per unit of time in the long run; it is a measure of service-capacity-oriented software availability. The steady-state reward-loss rate (or simply, reward loss), $1 - Y$, represents the relative amount of useful service lost per unit of time because of software failures. If we consider a specific group of failures in the analysis, the reward loss quantifies the service loss incurred by this group of failures.

The results of analysis are given in Table 4.5. The table shows the estimated reward loss incurred by software and nonsoftware failures, with SFT and NSFT. The bottom row of the table shows the improvement in service time (i.e., decrease in reward loss) achieved by providing the fault tolerance. The single-failure tolerance of the measured system reduces the service loss incurred by software failures by 89%, which clearly demonstrates the effectiveness of the single-failure tolerance of the measured system against software faults and corroborates the results obtained in the previous section. The table also shows that the single-failure tolerance reduces the service loss incurred by nonsoftware failures by 92% and that software problems account for 30% of the service loss in the measured system (with SFT).

A census of Tandem system availability [1] has shown that, as the reliability of hardware and maintenance improves significantly, software is the major source (62%) of outages in the Tandem system. It is inappropriate, however, to directly compare our number

Table 4.5 Estimated Loss of Service

Measure		Software	Nonsoftware	All
NSFT	$1-Y$	.00062	.00205	.00267
	Percent	23.2	76.8	100
SFT	$1-Y$	.00007	.00016	.00023
	Percent	30.4	69.6	100
Improvement		89%	92%	91%

(30%) with Gray's, because Gray's is an aggregate of many systems and ours is a measurement of a single system. Besides, the sources of the data and analysis procedures are different. Since the analysis performed in this section is based on automatically generated event logs, some nonsoftware problems requiring the replacement of faulty hardware can result in long recovery times and therefore great reward loss. Also, because of the experimental nature of the measured system, nonsoftware problems caused by operational or environmental faults may have been exaggerated. An operational or environmental fault can potentially affect all processors in the system.

4.3 Summary

This chapter evaluates the software fault tolerance of process pairs in the Tandem GUARDIAN system using human-generated software failure reports and on-line processor halt logs automatically generated by the operating system. The results of an evaluation using software failure reports showed that hardware fault tolerance buys software fault tolerance: process pairs in Tandem systems tolerate about 70% of reported field faults in the system software that cause processor failures. This result shows that, in a distributed transaction-processing environment, a significant level of software fault tolerance can be achieved by the use of checkpointing and restart, a technique for tolerating hardware faults. The loose coupling between processors, which results in the backup execution (the processor state and the sequence of events occurring) being different from the original execution, is a major reason for the measured software fault tolerance.

The results indicated that the actual level of software fault tolerance achieved by the use of checkpointing and restart depends on the degree of difference in the processing environment between the original execution and restart and on the proportion of subtle faults in software. While process pairs may not provide perfect software fault tolerance, the implementation of process pairs is not as prohibitively expensive as is developing and maintaining multiple versions of large software programs.

The results of Markov reward analysis using processor halt logs showed that the single-failure tolerance of the measured system reduces the service loss incurred by software failures by 89%. This result corroborates the results obtained using software failure reports. The results also showed that single-failure tolerance reduces the service loss incurred by nonsoftware failures by 92% and that software failures account for 30% of the service loss in the measured system (with SFT).

Chapter 5

Analysis of Failure Dependency

Failure dependency is a serious concern in parallel and fault-tolerant systems. Failure dependencies between two components can be analyzed using correlation coefficients. However, such dependencies can exist among multiple components. This chapter presents a method for analyzing multiway failure dependencies among software and hardware modules [43]. The method is developed based on multivariate statistical techniques, such as factor analysis and cluster analysis. The method is illustrated using the on-line processor halt log collected from an in-house Tandem VLX system for seven months (Chapter 2).

5.1 Processor Halt Matrix

The measured period was first divided into n equal intervals of 30 minutes each. Next, an $(n \times 8)$ matrix, termed the *processor halt matrix*, was constructed using the processor halt log. Both software and nonsoftware failures were used to build the matrix. The element (i, j) of this matrix has a value of 1, if processor j halts during the i-th time interval; otherwise, it has the value of 0. The processor halt matrix can be regarded as n samples of eight random variables. The j-th column of the matrix represents the sample halt history of processor j, while the i-th row of the matrix represents the state of the eight processors in the i-th time interval.

5.2 Correlation Analysis

The elements of the processor halt matrix were treated as real numbers, and the correlation between each pair of the random variables was calculated. Table 5.1 shows the resulting correlation matrix. Element (i, j) of the correlation matrix represents the halt correlation between processors i and j. Most of the correlations are low, but eight correlation coefficients are greater than 0.2 and four are greater than 0.7. These correlations require further investigation, because analysis shows that even low correlations can have a significant impact on system unavailability [44].

Table 5.1 Processor Halt Correlation

Processor	0	1	2	3	4	5	6	7
0	1.0	.	.	.	.	.	.	.
1	.0	1.0	.	.	.	.	.	.
2	-.00	.0	1.0	.	.	.	.	.
3	-.00	.0	-.00	1.0	.	.	.	.
4	.97	.0	.22	-.00	1.0	.	.	.
5	-.00	.0	-.00	.45	-.00	1.0	.	.
6	.0	.0	.77	-.00	.77	-.00	1.0	.
7	.0	.0	-.00	.77	-.00	.36	.40	1.0

5.3 Factor Analysis

The limitation of correlation analysis is that a correlation coefficient can quantify only the relationship between two variables. To investigate the multiway failure dependencies among the processors, factor analysis was performed. Factor analysis uncovers multiway statistical relationships among the observed variables by finding a set of underlying factors that link these variables [45]. For example, in a distributed system, a disk crash can account for the failures of those machines whose operations depend on a set of critical data on the disk. The disk state can be considered as a common factor of machine failures.

Let $X = (x_1, \ldots, x_p)^T$ be a vector of normalized, observable random variables. The factor model assumes

$$X = \Lambda F + E , \tag{5.1}$$

where $\Lambda = (\lambda_{ij})$ ($i = 1, ..., p$; $j = 1, ..., k$) is a matrix of constants called *factor loadings*, $F = (f_1, ..., f_k)^T$ is a random vector of elements called *common factors*, and $E = (e_1, ..., e_p)^T$ is a random vector of elements called *unique factors.* It is assumed that all common and unique factors are uncorrelated with each other, common factors are normalized, and unique factors have zero means. The variance of x_i can be expressed as

$$Variance(x_i) = 1 = \sum_{j=1}^{k} \lambda_{ij}^2 + \psi_i , \tag{5.2}$$

where ψ_i is the variance of e_i. The first term in the extreme right side of Equation (5.2), referred to as *communality*, is the amount of variance of x_i that is shared with other variables through common factors, while the second term is the amount of variance due to the unique variation of x_i. The correlation coefficient between x_i and f_j, which is λ_{ij}, represents the extent to which x_i depends on common factor f_j, and λ_{ij}^2 represents the amount of variance of x_i accounted for by common factor f_j.

Recall that the processor halt matrix is regarded as a collection of samples of eight random variables. These random variables can be represented by a random vector $X = (x_1, ..., x_8)^T$, where x_i corresponds to the halt behavior of processor i.

The SAS procedure FACTOR was used to perform factor analysis [46]. The results are shown in Table 5.2. Four common factors were identified. The matrix in the middle of the table is the Λ matrix, and the last column shows communality. The last two rows show the amount of variance explained by the common factors and their percentages of the total variance.

According to [45], factor loadings greater than 0.5 are considered significant. However, in a reliability analysis, factor loadings lower than 0.5 can be significant. Processor 1 showed independent behavior. Its contribution to the common factors, as measured by the communality, is zero. The behavior of processor 1 presents a unique factor. Closer examination showed that processor 1 experienced no halts. Common factor 2 captures

Table 5.2 Factor Patterns of Processor Halt

Processor	Common Factor 1	Common Factor 2	Common Factor 3	Common Factor 4	Communality
0	.997	-.004	-.069	.023	1.0
1	.0	.0	.0	.0	.0
2	.061	.012	.853	-.133	.75
3	.001	.999	-.011	.021	1.0
4	.982	-.000	.188	-.018	1.0
5	-.001	.447	-.005	.009	.20
6	.047	-.002	.862	.506	1.0
7	-.007	.762	.090	.641	1.0
Variance	1.965	1.781	1.519	.685	
Fraction (%)	24.6	22.3	19.0	8.6	

the multiway dependency among processors 3, 5, and 7, although the contribution of processor 5 is small (0.447^2, i.e., 20% of its variance is explained by this factor). Common factor 3 captures the multiway dependency among processors 2, 4, and 6. Again, the contribution of processor 4 is small (0.188^2, i.e., 3.5% of its variance is explained by this factor).

Partly because of the experimental nature of the measured system, and partly because on-line processor halt logs do not provide the information of underlying causes of processor halts, it is not easy to make accurate interpretations of the common factors. We speculate that common factors 2 and 3 are attributed to the failures of process pairs caused by software faults and experiments related to the development of software running on multiple processors. Processors 0 and 1, processors 2 and 3, and processors 4 and 5 of the measured system shared dual-port disk controllers. However, the results of factor analysis show that this architectural coupling did not cause halt dependencies between the processors.

5.4 Cluster Analysis

Although factor analysis is a valuable technique for characterizing multiway dependencies, it does not provide detailed information on patterns which relate the measured variables. To investigate how the processors in the measured system halted, a statistical clustering technique was used.

Each row of the processor halt matrix (the state of the eight processors in each time interval), which is referred to as an *observation*, was regarded as a point in an eight-dimensional space. The distance between two observations was defined as the number of mismatches between pairs of elements in the same columns of the corresponding two rows. Only the observations in which at least one processor halted were clustered.

The FASTCLUS procedure of the SAS software package was used to perform statistical clustering [46]. FASTCLUS is based on the k-means algorithm, a popular non-hierarchical clustering technique, which groups observations into k nonempty clusters (C_1, C_2, ..., C_k) that minimize the total sum of the squares of the Euclidean distances of the cluster members from their centroids [47]:

$$D = \sum_{j=1}^{k} \sum_{x_i \in C_j} \| x_i - \overline{x_j} \|^2 . \tag{5.3}$$

In the equation, x_i is the i-th observation, and $\overline{x_j}$ is the centroid of cluster C_j, i.e.,

$$\overline{x_j} = \frac{1}{m_{C_j}} \sum_{x_i \in C_j} x_i , \tag{5.4}$$

where m_{C_j} is the number of observations in cluster C_j. In this analysis, the distance between the two observations is

$$\| x_i - x_l \| = \sum_{n=1}^{8} (x_{in} \oplus x_{ln}) , \tag{5.5}$$

where $\oplus$ represents the exclusive-OR operation. Note that x_i represents the i-th row of the processor halt matrix in cluster analysis. In the factor analysis discussed in the previous section, x_i represented the i-th column of the matrix. This difference allows cluster analysis to identify the dependency patterns.

Table 5.3 summarizes the results of cluster analysis. The cluster centroids, which are the column vectors in the table, represent the processor halt patterns in the measured system. Each element of a cluster centroid represents the probability that the corresponding processor will halt in that pattern. For example, in the pattern represented by cluster 1, processor 0 always halted, while processor 1 never halted. Clusters 4 and 5 capture the patterns which involve three processors. There were three time intervals in which processors 2, 4, and 6 halted, and there were another three time intervals in which processors 3, 5, and 7 halted. Note that the cluster analysis collaborates with the factor analysis: clusters 1, 3, 4, and 5 capture the patterns indicated by common factors 1, 4, 3, and 2, respectively, while cluster 2 captures the independent halts of processor 5 as indicated by its low communality in Table 5.2.

Table 5.3 Centroids of Processor Halt Clusters

Processor	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5
0	1.0	0	0	0	0
1	0	0	0	0	0
2	0	0.08	0	1.0	0
3	0	0	0	0	1.0
4	1.0	0	0	1.0	0
5	0	0.92	0	0	1.0
6	0	0	1.0	1.0	0
7	0	0	1.0	0	1.0
#Observation	42	13	2	3	3
Fraction (%)	66.7	20.6	3.2	4.8	4.8

5.5 Summary

This chapter presents a method for analyzing multiway failure dependencies among software and hardware modules. The method was developed based on multivariate statistical techniques, such as factor analysis and cluster analysis. An illustration of the method using processor halt logs demonstrated that factor analysis can unearth the underlying multiway failure dependencies and that cluster analysis can identify the actual dependency patterns.

Chapter 6

Reliability Modeling of Operational Software

Software reliability models attempt to estimate the reliability of software. Many models have been proposed [9], [48]. These models typically attempt to relate the history of fault identification during the development phase, verification efforts, and operational profile. The primary focus is on the software development process, and the underlying assumptions are that software is an independent entity and each software fault has the same impact.

The results from the previous chapters indicated that there are other factors that significantly impact the dependability of operational software. First, there are both highly visible and less visible software faults. A single, highly visible software fault can cause many field failures, and recurrences can seriously degrade software dependability in the field. Second, for a class of software such as operating systems, the fault tolerance of the overall system can significantly improve software dependability by making the effects of software faults invisible to users. Clearly, dependability issues for operational software in general can be quite different from those for the software in the development phase.

This chapter asks the question: which factors determine the dependability of the measured operating system? Using the software failure and recovery characteristics identified in the previous chapters, this chapter builds a model that describes the impact of faults in the Tandem GUARDIAN operating system on the reliability of an overall Tandem system in the field. The model is used to evaluate the significance of the factors considered and to identify areas where improvement efforts can be directed by conducting sensitivity analysis [49].

6.1 Model Construction

We considered a hypothetical Tandem system whose software reliability characteristics are described by the parameters in Table 6.1. In this analysis, the term *software reliability* means the reliability of an overall system when only the faults in the system software are considered. All parameters in the table except λ and μ were estimated based on the measured data (Section 3.1 and Chapter 4). The values of λ and μ were determined to mimic the 30 years of software mean time between failure (MTBF), and the mean time to repair (MTTR) characteristics reported in [1]. Thus, the objectives of the analysis were to model and evaluate reliability sensitivity to various factors, not to estimate the absolute software reliability.

Table 6.1 Estimated Software Reliability Parameters

Failures:	First Occurrence	Recurrence	Unidentified
Failure rate	$\lambda_f = 0.24\lambda$	$\lambda_r = 0.61\lambda$	$\lambda_u = 0.15\lambda$
Prob(double CPU halt|software failure)	$C_{df} = 0.23$	$C_{dr} = 0.18$	$C_{du} = 0.0$
Prob(system failure|double CPU halt)	$C_{sdf} = 0.44$	$C_{sdr} = 0.63$	$C_{sdu} = 0.0$
Prob(system failure due to severe, single CPU halt)	$C_{ssf} = 0.05$	$C_{ssr} = 0.0$	$C_{ssu} = 0.0$
Failures: Software failure rate = λ = 0.32/year			
Recovery: Recovery rate = μ = 1/hour			

In Table 6.1, "Prob(double CPU halt|software failure)" is the probability that a double processor halt (i.e., the failure of a process pair) occurs given that a software failure occurs. Similarly, "Prob(system failure|double CPU halt)" is the probability that a system failure occurs given that a double processor halt occurs. In this analysis, a system failure was defined to occur when more than half of the processors in the system failed. These two parameters are used to describe the major failure mode of the system because of software. The parameter "Prob(system failure due to severe, single CPU halts)" represents the secondary failure mode, which captures single processor halts severe enough to cause system coldloads. The table shows these probabilities for first occurrences, recurrences, and unidentified failures.

Based on the parameters in Table 6.1 and on the following assumption, we built a continuous-time Markov model to describe the software failure and recovery in a hypothetical eight-processor Tandem system in the field.

> Assumption 1: The time between software failures in the system has an exponential distribution, and the three types of failures (first occurrence, recurrence, and unidentified) are randomly mixed.

This assumption was necessary, because determining the above characteristics for a single system would require a minimum of a few hundred years of measurements. The assumption could not be validated using the measured data, because the measured data was collected from a large number of user systems running different versions of the operating system and having different operational environments and system configurations. Given this situation, the assumption seemed a reasonable choice.

Figure 6.1 shows the Markov model. In the model $S_i, i = 0, .., 4$ represents that i processors are halted because of software faults. A system failure is represented by the S_{down} state. To evaluate software reliability, no recovery from a system failure was assumed. That is, the system failure state is an absorption state. The R_i state represents an intermediate state in which the system tries to recover from an additional software failure (i-th processor halt) using process pairs.

If a software failure occurs during the normal system operation (i.e., when the system is in the S_0 state), the system enters the R_1 state. If the failure is severe enough to cause a system coldload, a system failure occurs; otherwise, the system attempts to recover from the failure by using backups. If recovery is successful, the system enters the S_1 state; otherwise, a double processor halt occurs. If the two halted processors control key system resources (such as a set of disks) that are essential for system operation, the rest of the processors in the system also halt and a system failure occurs; otherwise, the system enters the S_2 state and continues to operate. The value of r, the transition rate

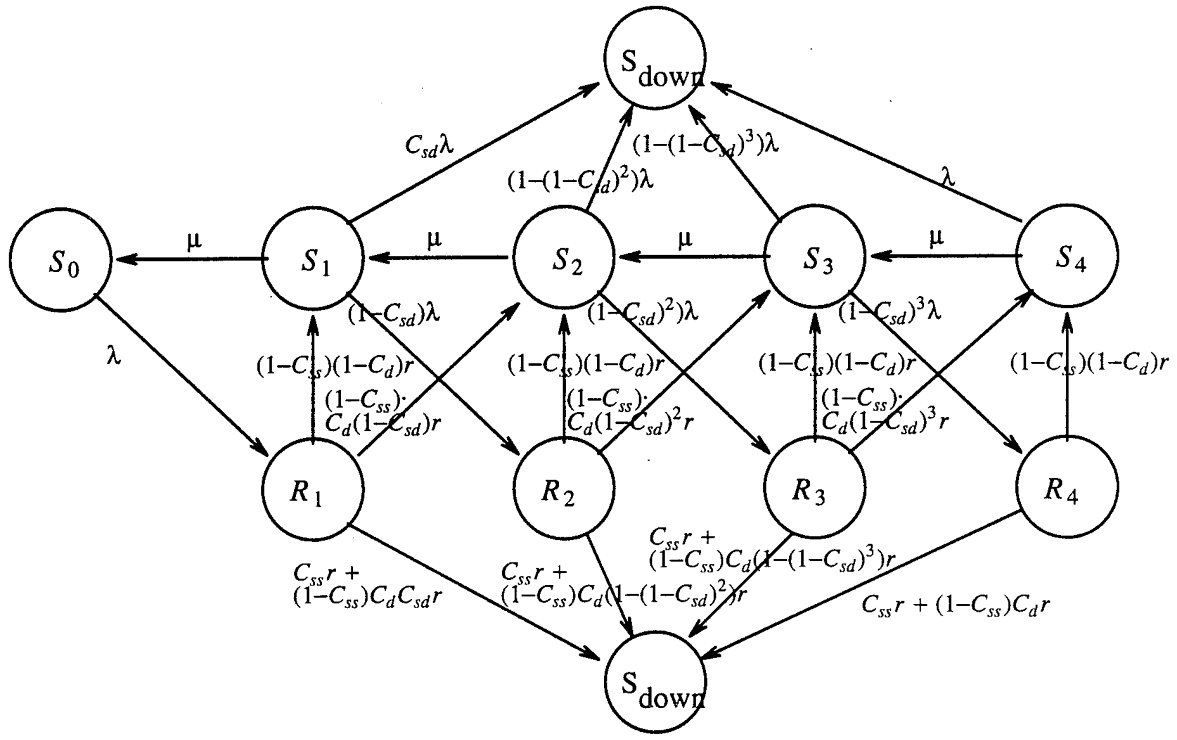

S_down
$C_{sd}\lambda$
$(1-(1-C_{sd})^3)\lambda$
$(1-(1-C_{sd})^2)\lambda$
λ
S_0
μ
S_1
μ
S_2
μ
S_3
μ
S_4
λ
$(1-C_{sd})\lambda$
$(1-C_{sd})^2\lambda$
$(1-C_{sd})^3\lambda$
$(1-C_{ss})(1-C_d)r$
$(1-C_{ss})C_d(1-C_{sd})r$
$(1-C_{ss})(1-C_d)r$
$(1-C_{ss})C_d(1-C_{sd})^2r$
$(1-C_{ss})(1-C_d)r$
$(1-C_{ss})C_d(1-C_{sd})^3r$
$(1-C_{ss})(1-C_d)r$
R_1
R_2
R_3
R_4
$C_{ss}r + (1-C_{ss})C_dC_{sd}r$
$C_{ss}r + (1-C_{ss})C_d(1-(1-C_{sd})^2)r$
$C_{ss}r + (1-C_{ss})C_d(1-(1-C_{sd})^3)r$
$C_{ss}r + (1-C_{ss})C_dr$
S_down

Figure 6.1 Software Reliability Model

out of an R_i, is small and has virtually no impact on software reliability; a value of one transition per minute was used in the analysis. Since the system stays in an R_i state for a short time, additional failures occurring in an R_i state were ignored; in fact, these failures are implied in the failure rate (λ) in the corresponding S_i and S_{i+1} states. Given the model in Figure 6.1, software reliability of the system can be estimated by calculating the distribution of time for the system to be absorbed to the S_{down} state, starting from the S_0 state.

In Figure 6.1, the three coverage parameters C_d, C_{sd}, and C_{ss} were calculated from Table 6.1:

$$C_d = \text{Prob.(double CPU halt|software failure)} = \frac{\lambda_f C_{df} + \lambda_r C_{dr} + \lambda_u C_{du}}{\lambda_f + \lambda_r + \lambda_u}, \qquad (6.1)$$

$$C_{sd} = \text{Prob.(system failure|double CPU halt)} = \frac{\lambda_f C_{df} C_{sdf} + \lambda_r C_{dr} C_{sdr} + \lambda_u C_{du} C_{sur}}{\lambda_f C_{df} + \lambda_r C_{dr} + \lambda_u C_{du}}, \quad (6.2)$$

and

$$C_{ss} = \text{Prob.(system failure due to severe, single CPU halts)} = \frac{\lambda_f C_{ssf} + \lambda_r C_{ssr} + \lambda_u C_{ssu}}{\lambda_f + \lambda_r + \lambda_u}. \tag{6.3}$$

The parameter C_d includes the two cases explained in Section 4.1: the failure of a process pair caused by a single software fault, and the failure of a process pair caused by two software faults (the second halt occurs during job takeover). The parameter C_{sd} represents the probability that the system loses a set of disks as a result of a double processor halt. The parameter C_{sd} is determined primarily by the system configuration and is discussed further in Section 6.4. The above three parameters can actually be obtained directly from Table 4.4 in Subsection 4.1.5. Equations (6.1), (6.2) and (6.3) will be used to investigate the impact of recurrences (λ_r) on software reliability in Section 6.2.

The model (Figure 6.1) includes the effect of multiple independent software failures. For example, if a software failure occurs when the system is in the S_i state ($i \neq 0$), the following three system failure scenarios must be considered: the system fails regardless of whether the new failure causes a single or double processor halt, because the first processor halted because of the new failure causes a set of disks to become inaccessible; the system fails because the new failure is severe and can only be recovered by a system coldload; and the new software failure causes a double processor halt, and the second processor halted causes a set of disks to become inaccessible. This situation is shown in Figure 6.2.

It was not possible to estimate the branching probabilities in Figure 6.2 for each state from the data, because the major failure mode (i.e., a software failure occurring when the system is in the S_0 state causes a double processor halt and subsequently causes a system failure) was dominant. To estimate these parameters, the following assumption was made.

> Assumption 2: The probability that any two processors in the system control a set of disks that is essential for system operation is equal to the parameter C_{sd}, measured using failures from all user systems.

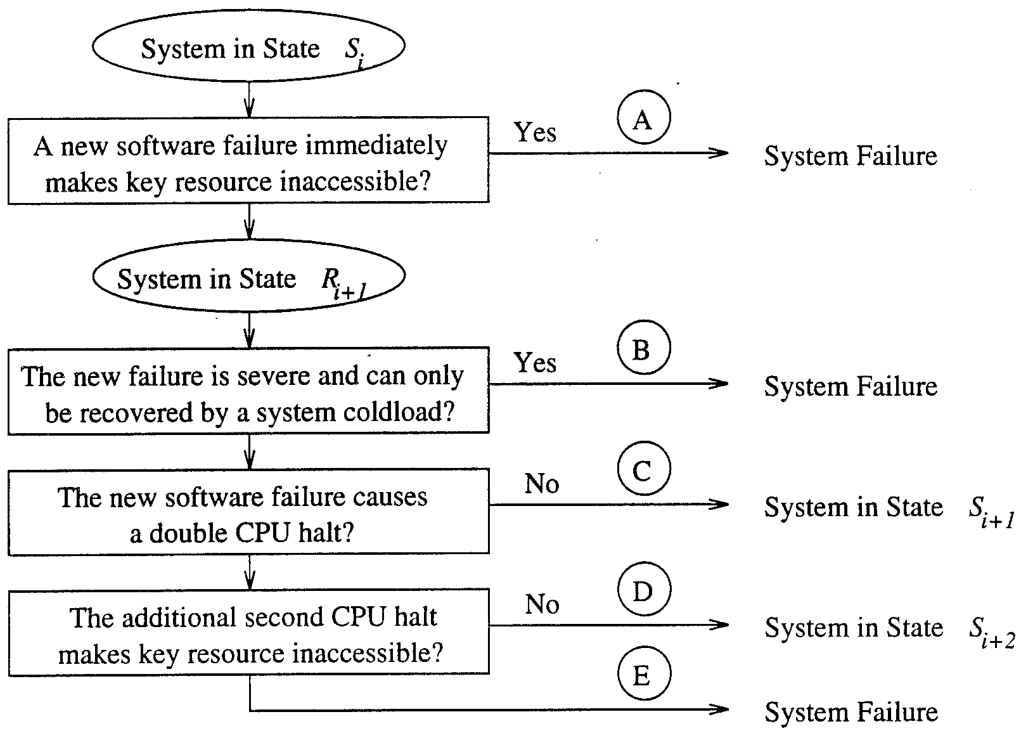

System in State S_i
A new software failure immediately makes key resource inaccessible?
Yes
A
System Failure
System in State R_{i+1}
The new failure is severe and can only be recovered by a system coldload?
Yes
B
System Failure
The new software failure causes a double CPU halt?
No
C
System in State S_{i+1}
The additional second CPU halt makes key resource inaccessible?
No
D
System in State S_{i+2}
E
System Failure

Figure 6.2 Effect of Multiple Independent Software Failures

Again, this assumption was necessary because C_{sd} for a single system could not be determined from the measured data. A value of C_{sd} measured using failures from all user systems would be a reasonable estimate.

Table 6.2 shows the branching probabilities in Figure 6.2 calculated for each S_i $(i \neq 0)$ state. For example, given that an additional software failure causes a double processor halt when the system is in the S_1 state, the probability that the third processor halt does not cause a system failure (path D in Figure 6.2) is $(1 - C_{sd})^2$ because the probability that the third processor halted and either of the two processors which were already halted control a set of disks (i.e., cause a system failure) is C_{sd}. The branching probabilities in Table 6.2 were used to determine the corresponding transition rates in the model (Figure 6.1).

The same recovery rate was used regardless of the number of processors halted, because the recovery time is typically determined by the time required to take a memory dump, and a memory dump is taken from one processor at a time. Previous studies

Table 6.2 Parameters for Multiple Independent Software Failures

	$i=1$ (i.e., system in S_1)	$i=2$	$i=3$	$i=4$
Path A	C_{sd}	$1-(1-C_{sd})^2$	$1-(1-C_{sd})^3$	1
Path B	$(1-C_{sd})C_{ss}$	$(1-C_{sd})^2C_{ss}$	$(1-C_{sd})^3C_{ss}$	–
Path C	$(1-C_{sd})(1-C_{ss})\cdot(1-C_d)$	$(1-C_{sd})^2(1-C_{ss})\cdot(1-C_d)$	$(1-C_{sd})^3(1-C_{ss})\cdot(1-C_d)$	–
Path D	$(1-C_{sd})(1-C_{ss})\cdot C_d(1-C_{sd})^2$	$(1-C_{sd})^2(1-C_{ss})\cdot C_d(1-C_{sd})^3$	–	–
Path E	$(1-C_{sd})(1-C_{ss})\cdot C_d(1-(1-C_{sd})^2)$	$(1-C_{sd})^2(1-C_{ss})\cdot C_d(1-(1-C_{sd})^3)$	$(1-C_{sd})^3(1-C_{ss})C_d$	–

assumed that the failure rate is proportional to the number of processors up and working [50]. The same software failure rate was assumed in all states, considering that, as more processors halt, the rest of the processors will receive more stress. Again, the dominance of the major system failure mode did not allow us to estimate the parameters from the data.

The distribution of time for the system to be absorbed to the system failure state, starting from the normal state, was evaluated using the model in Figure 6.1. SHARPE [51] was used for the evaluation. Figure 6.3 shows the software reliability curve of the modeled system and confirms the assumed software MTBF of 30 years. The figure represents the reliability of an overall Tandem system in the field when only the faults in the system software are considered.

6.2 Reliability Sensitivity Analysis

Table 6.3 shows six factors considered in the analysis. The second column of the table shows activities related to these factors, and the third column shows the model parameters affected by the factors. For example, a 10% reduction in the number of faults that cause software failures, which can be achieved by improving the software development process, will reduce λ by 10% by reducing λ_f, λ_r and λ_u simultaneously. (Note that the number of faults identified can be regarded as the software failure rate

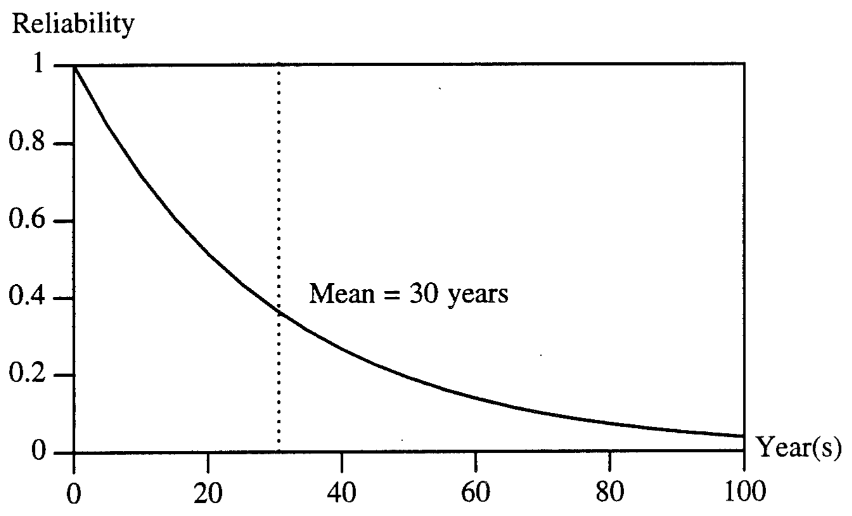


Figure 6.3 System Reliability Due to Software

when only a single copy of the software runs.) On the other hand, a 10% reduction in the recurrence rate (λ_r), which can be achieved by improving the software service process, will reduce λ by 6.1% (Table 6.1) and change C_d, C_{sd}, and C_{ss} accordingly. Refer to Equations (6.1), (6.2) and (6.3).

Table 6.3 Factors of Software Reliability

Factor	Activity	Related Parameters	
		Detailed	Overall
Number of faults	Software development	λ_f, λ_r, λ_u	λ
Recurrence rate	Software service	λ_r	λ, C_d, C_{sd}
Coverage parameter C_d	Robustness of process pairs	C_{df}, C_{dr}, C_{du}	C_d
Coverage parameter C_{sd}	System configuration	C_{sdf}, C_{sdr}, C_{sdu}	C_{sd}
Coverage parameter C_{ss}	-	C_{ssf}, C_{ssr}, C_{ssu}	C_{ss}
Recovery time	Diagnosability/maintainability	μ	μ

The coverage parameters C_d and C_{sd} are determined primarily by the robustness of process pairs and the system configuration, respectively. For example, C_d can be reduced by conducting extra testing of the routines related to job takeover. The parameter C_{sd} is determined by the location of failed process pairs and the disk subsystem configuration. This parameter is discussed further in Section 6.4. Analytical models for predicting coverage in a fault-tolerant system and the sensitivity of system reliability/availability to the coverage parameter were discussed in [52]. The recovery rate μ can be improved by automating the data collection and reintegration process.

Figure 6.4 shows the software MTBF evaluated using the model in Figure 6.1 while varying the six factors in Table 6.3, one at a time. It is interesting to see that C_d and C_{sd} are almost as important as λ in determining the software MTBF. For example, a 20% reduction in C_d or C_{sd} has as much impact on software MTBF as an 18% reduction in λ. (The figure shows that the impact is approximately a 20% increase in software MTBF.) This result is understandable because the system fails primarily because of a double processor halt causing a set of disks to become inaccessible, not because of multiple independent software failures.

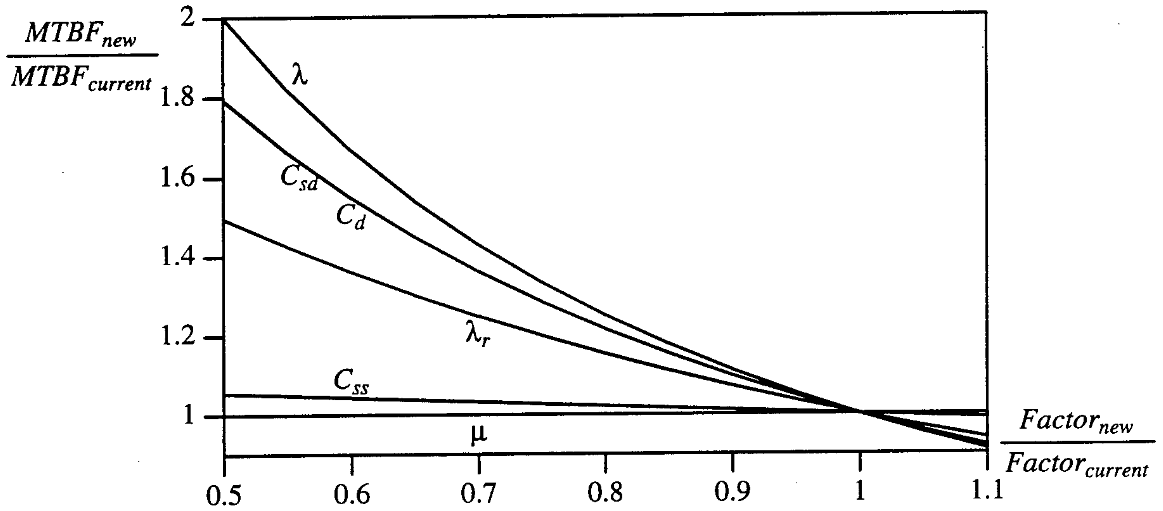


Figure 6.4 Software MTBF Sensitivity

Figure 6.4 also shows that the recurrence rate has a significant impact on software reliability. A complete elimination of recurrences ($\lambda_r = 0$ in Table 6.1) would increase the software MTBF by a factor of three. The impact of C_{ss} on software reliability is small, because severe, single processor halts causing system coldloads are rare. The impact of μ on software MTBF is virtually nil. In other words, recovery rate is not a factor as far as software reliability is concerned, again, because the system is unlikely to fail because of multiple independent software failures.

Typically, it is assumed that the number of faults in software is the only major factor determining software reliability. Figure 6.4 clearly shows that in the Tandem system, there are four degrees of freedom in improving the software reliability: the number of faults in software, the recurrence rate, the robustness of process pairs, and the system configuration strategy. The first two are general factors, and the last two are platform-dependent factors. Efforts to improve software reliability can be optimized by estimating the cost of improving each of the four factors.

6.3 Reliability Sensitivity to Fault Category

This section investigates the impact of software faults in different fault categories (Table 3.2 in Section 3.1) on software reliability. In this section, a *failure group* is defined as the group of software failures caused by all faults that belong to a fault category. We estimated the software MTBF by assuming that each failure group is empty, i.e., the faults in a fault category did not cause software failures. The failure rate and the coverage parameters for the model in Figure 6.1 were adjusted:

$$\lambda_{new} = \frac{\text{total no. of software failures} - \text{no. of software failures in a failure group}}{\text{total no. of software failures}} \lambda , \qquad (6.4)$$

$$C_d = \frac{\text{total no. of double CPU halts} - \text{no. of double CPU halts in a failure group}}{\text{total no. of software failures} - \text{no. of software failures in a failure group}} , \qquad (6.5)$$

$$C_{sd} = \frac{\text{total no. of system failures} - \text{no. of system failures in a failure group}}{\text{total no. of double CPU halts} - \text{no. of double CPU halts in a failure group}} , \qquad (6.6)$$

and

$$C_{ss} = \frac{\text{total no. of severe, single CPU halts} - \text{no. of severe, single CPU halts in a failure group}}{\text{total no. of software failures} - \text{no. of software failures in a failure group}} . \qquad (6.7)$$

In Equation (6.6), only those system failures caused by double processor halts (i.e., failures of process pairs) were counted.

Table 6.4 shows the results. The last column of the table shows the improvement in software MTBF when failures caused by each fault category are eliminated. Only those categories that have more than ten failures were considered. The table shows that "Missing operation" caused the greatest reliability loss. Further analysis showed that uninitialized pointers (Table 3.2 in Section 3.1) were responsible for more than half of this loss. The table also shows that "Unexpected situation" is another significant source of reliability loss. Most of this loss is attributed to faults such as incorrect parameters passed by user processes, illegal procedure calls made by user processes, and not considering all legitimate operational scenarios in designing software. (The reliability loss is not attributed to subtle faults such as race conditions and timing problems.) Additional code inspection and testing efforts can be directed to these fault categories. Unidentified failures had virtually no impact on software reliability, because all of these failures caused single processor halts.

Table 6.4 Reliability Sensitivity to Fault Category

Fault Category	#Failures	$\frac{MTBF_{improved}}{MTBF_{current}}$
Incorrect computation	3	-
Data fault	21	1.00
Data definition fault	7	-
Missing operation	27	1.47
Side effect of code update	5	-
Unexpected situation	46	1.35
Microcode defect	8	-
Other	12	1.06
Unidentified	26	1.00
Unable to classify	24	1.12

6.4 Impact of System Configuration on Software Dependability

System configuration is an issue that demonstrates the importance of considering the interactions between hardware, software, and operations. Table 6.5 shows a breakdown of the process pairs whose failures caused the 18 observed system failures, based on their configurability. In the table, a “Location-free” process pair refers to a pair that can be placed on any two processors in the system, independent of hardware configuration. The location of a nondisk or disk I/O process pair is determined by hardware configuration. The failure of a nondisk I/O or location-free process pair causes a system failure, because the process pair executes on the two processors that execute a disk process pair. Thus, a double processor halt resulting from a failure of such a nondisk I/O or location-free process pair would cause a set of disks to become inaccessible. Table 6.5 shows that the number of system failures could potentially be reduced by 67% (12 out of 18) by avoidance of the overlap in location between disk process pairs and the failed nondisk or location-free process pairs. This result demonstrates the importance of considering software dependability in the context of an overall system.

Table 6.5 Configurability of Failed Process Pairs That Caused System Failures

Failed Process Pair	#System Failures
Location-free process pair	7
Non-disk I/O	5
Disk I/O process pair	2
Others	4

6.5 Summary

This chapter seeks to identify factors determining the dependability of operational software. We built a software reliability model that describes the impact of software faults on an overall Tandem system in the field and conducted reliability sensitivity analysis using the model. The results showed that, in addition to the conventional approach of reducing the number of faults in software, software dependability in Tandem systems can be enhanced by reducing the recurrence rate and by improving the robustness of process pairs and the system configuration. The number of faults in software and the recurrence rate are general factors; the robustness of process pairs and the system configuration are platform-dependent factors. Improvement efforts can be optimized by estimating the cost of improving each factor.

The results also showed that the impact of software fault tolerance (i.e., the robustness of process pairs) and the impact of system configuration are as significant as is the impact of the number of faults in software. A complete elimination of recurrences would triple the software MTBF. The analysis of reliability sensitivity to fault category showed that faults such as "Missing operation" and "Unexpected situation" are the major causes of software reliability loss. Additional code inspection and testing efforts can be directed to these types of faults. The investigation of the impact of system configuration on software dependability demonstrated the importance of considering interactions between software and hardware in the context of an overall system.

Chapter 7

Diagnosis of Recurrences

After all analyses of software dependability, one issue stands out. This issue is recurrence. This chapter presents a system-independent approach for automatically diagnosing recurrences based on their symptoms [53]. Characteristics of recurrences are analyzed first: their causes, effects, and symptoms are investigated. Based on the results of this analysis, an approach for automatically diagnosing recurrences based on symptoms is developed: a diagnosis environment is discussed, a diagnosis strategy (i.e., a set of symptoms and an associated matching scheme for diagnosing recurrences) is proposed, and a method for evaluating the effectiveness of a diagnosis strategy is presented. The effectiveness of the proposed diagnosis strategy is evaluated using actual failure and repair data collected from two Tandem system software products. Recurrences exist at all levels of software problems. This chapter focuses on processor failures caused by faults in the system software.

7.1 Analysis of Recurrences

The results in Chapter 3 show that 72% of reported field software failures in Tandem systems are recurrences. Recurrences are not unique to Tandem systems. A similar situation exists in IBM systems [54] and AT&T systems [55]. Clearly, in environments where many users run the same software, the number of faults in software is not the only factor determining software dependability. Recurrences can seriously degrade software dependability in the field.

Recurrences exist for several reasons. First, designing and testing a fix of a problem can take a significant amount of time. In the meantime, recurrences can occur at the same site or at other sites. Second, the installation of a fix sometimes requires a planned outage, which may force users to postpone the installation and thus cause recurrences. Third, a purported fix can fail. Finally and probably most importantly, users who did not experience problems due to a certain fault often hesitate to install an available fix for fear that doing so will cause new problems.

The effects of recurrences are: more failures than predicted based on the number of faults; wasted resources due to repeated data collection, reporting, and diagnosis of the same problem; and delayed service to users even if solutions to problems are available. An attempt was made to determine an optimal preventive service policy to address the first effect [54]. Preventive service is the process of fixing a software fault in a user system, even though the fault has not caused a problem in the system. Preventive service has the potential to reduce the number of recurrences, but it costs resources. Further, faults in a fix can cause new problems in user systems. Based on the failure and shipment data of IBM software products, the study proposed that preventive service be limited to a small number of highly visible faults. This result and the described reasons for recurrences indicate that recurrences will continue to be a significant part of field software failures.

7.1.1 Distributions of recurrences

Figure 7.1 shows the cumulative number of software failures reported during the measurement period for first occurrences, recurrences, unidentified failures, and all failures. For each type of failure, the hypothesis that the distribution is uniformly distributed could not be rejected by the Kolmogorov-Smirnov test at the 1% significance level. (The Kolmogorov-Smirnov test is commonly used to test the goodness of fit because it directly compares a raw distribution to a hypothesized distribution [38].) This result is reasonable, because the data was collected from all user systems where events such as software upgrade and the installation of fixes of known faults occur asynchronously. As a result, the aggregate distribution is expected to be close to a uniform distribution regardless of the type of the failure distribution in each site. This uniform occurrence of failures is a desirable characteristic from a system service point of view.

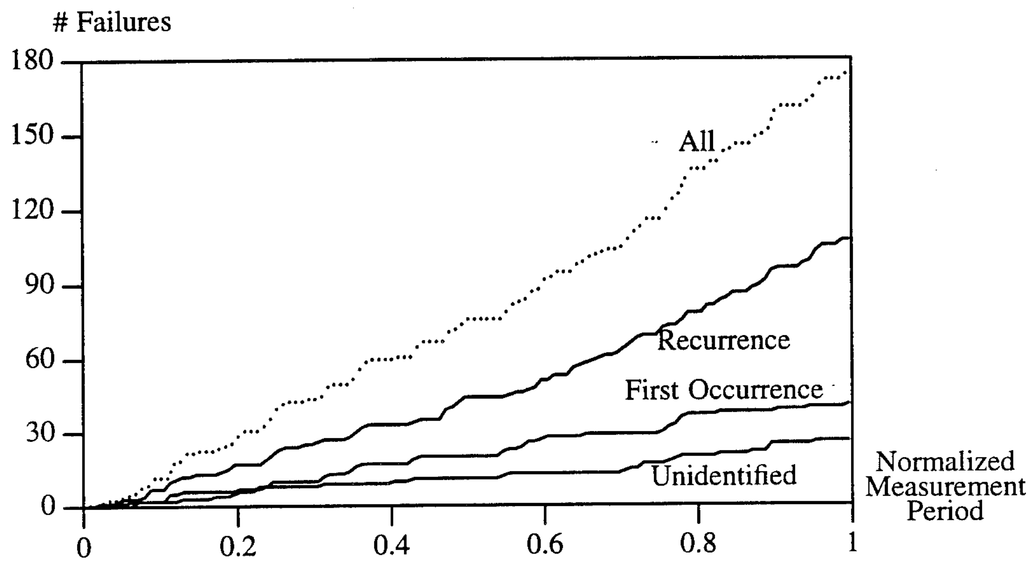


Figure 7.1 Cumulative Number of Software Failures by Failure Type

A first occurrence and a set of recurrences caused by the same software fault are referred to as a *recurrence group*. The 153 TPRs whose underlying software faults were identified formed 100 recurrence groups, which was expected, because there were 100 unique faults in Table 3.2. Typically, a TPR reporting a recurrence has pointers to the TPRs reporting the corresponding first occurrence and other recurrences. We searched the TPR database for all past TPRs sharing the same causes with the 100 recurrence groups using these pointers and expanded the size of the recurrence groups. After this expansion, 69 recurrence groups had more than one TPR.

Failures in each recurrence group were ordered in time, and the time between each two adjacent failures was measured. Figure 7.2 shows both the raw and fitted cumulative density functions for the time between recurrences, obtained using all instances from the 69 recurrence groups. The distribution is well represented by a two-phase hyperexponential distribution. The hypothesis that the distribution fits a simple exponential was rejected by the Kolmogorov-Smirnov test at the 5% significance level. But the hypothesis that the distribution fits a two-phase hyperexponential function could not be not rejected using the same test at the same level.

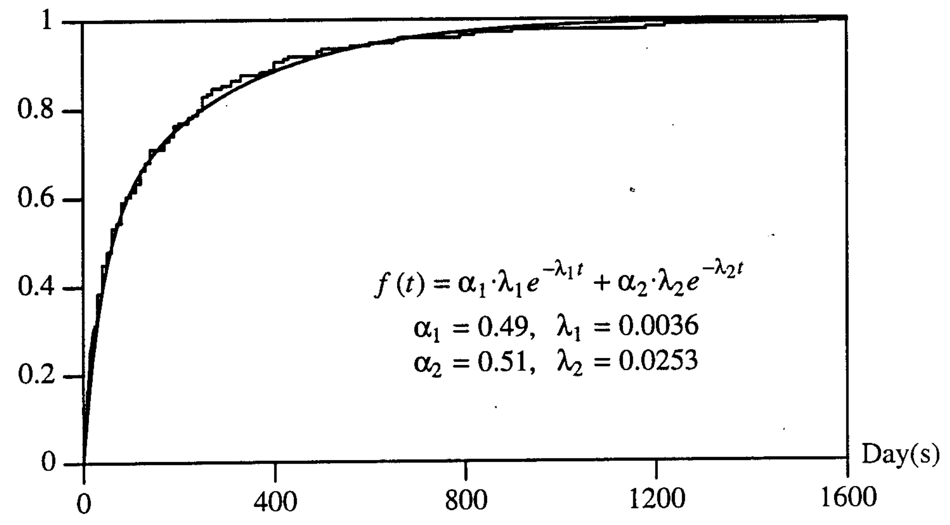


Figure 7.2 Cumulative Distribution of Time between Recurrences

Error-burst phenomena in a computer system have been reported by previous studies [42]. Our results show that a similar phenomenon exists in a much larger environment, i.e., when we look at operating system failures from a large number of user systems. In Figure 7.2, the recurrence-burst phenomenon is captured by the higher recurrence rate (λ_2—the average time between recurrences is $1/\lambda_2 \approx 40$ days) with a weight of 0.51. The lower rate (λ_1—the average time between recurrences is $1/\lambda_1 \approx 275$ days), with a weight of 0.49, captures the time between failure bursts in a recurrence group. Isolated recurrences were primarily because of installing a software version that contains a known fault that is almost forgotten and because of known faults that rarely expose themselves. A site can carry known faults that rarely expose themselves, because they are not likely to cause failures.

There were recurrence groups in which a burst of recurrences followed long after the first occurrence. This fault reappearance was primarily because of side effects of software maintenance (i.e., a code update causing a fixed fault to reappear). The data also showed that, although they are statistically insignificant, there are quick recurrences within a couple of days. Considering that quick recurrences in a site are typically reported as a single TPR, this quick recurrence could be a significant recurrence mode in actual environments. These recurrence modes indicate that recurrences can be reduced by providing quick service for a new fault, by ensuring that known and fixed faults are not reintroduced in the code or in user systems, and by better judging the set of sites that are vulnerable to a known fault.

7.1.2 Symptoms of recurrences

Failures in a recurrence group share the same underlying software fault, but their symptoms can differ. The investigation of error propagation in failures caused by the same software fault indicated that they often had identical stack traces, which suggested that an automatic diagnosis of recurrences based on symptoms may be possible. A stack trace is the history of procedure calls made by the process that was running at the time of the software failure.

Figure 7.3 shows a stack trace extracted from a failure. Each line represents a procedure, and the associated number represents the offset of the code location (i.e., the machine instruction) that called the next procedure from the beginning of the procedure, in octal words. In Figure 7.3, the system process that asserted the processor halt is normally sitting in the procedure MAINLOOP. When the process receives a request, it serves the request by calling the necessary procedures. In this case, the process detected a nonrecoverable error during the execution and halted the processor on which it was running. The set of procedures shown in Figure 7.3 constitutes a stack trace for the failure. Each software failure has its stack trace.

The first line from the top shows an error-handling procedure. There is an error-handling procedure and an associated halt code for each type of problem detection defined by software developers and system designers. In the sample shown in Figure 7.3, the error-handling procedure shows that a page fault occurred during the execution of a code section where a page fault is not supposed to occur. The actual stack trace consists of the procedure names beginning with the second line. The stack trace represents the

```
Halt code →  %00104 into PAGEFAULT
TOS →        %00365 into GET_ACCTENTRY
             %00220 into GET_FILEOP_INPUTBUF
             %00052 into READ_SETUP
             %00015 into READ_RQST
             %00446 into MAINLOOP
```

Figure 7.3 A Stack Trace with Offsets

software function that detected the problem, i.e., what the processor was doing at the time of failure. The stack trace is not necessarily related to the location of the underlying software fault.

To determine the significance of the observation that failures caused by the same software fault often have identical stack traces and to understand why failures caused by the same software fault have different stack traces, we compared the stack traces of failures in each recurrence group [34]. The first two columns of Table 7.1 show the results of matching the stack traces of failures in each recurrence group. The last two columns of the table show a further breakdown of each group based on the propagation modes discussed in Subsection 3.2.2. (Error latency was not considered here.) The table shows that in 70% of the recurrence groups (44 out of 63), all failures in a recurrence group have identical stack traces.

The third and fourth columns of Table 7.1 show that error propagation (i.e., "Further corruption") is a major reason that failures caused by the same software fault have different stack traces. Once many errors are generated because of error propagation, there can be many possible detection scenarios, and each of these can give a distinct stack trace. A further investigation showed that faulty functions in an operating system that can be called by many processes are a major reason that "Quick detection" and "No propagation" lead to different stack traces in a recurrence group. Table 7.1 also shows that "No propagation," which is significant because of the use of defensive programming techniques, helps increase the proportion of recurrence groups that have identical stack traces.

Table 7.1 Comparison of Stack Traces in Recurrence Groups

Results of Comparison	# Recurrence Groups	Propagation Mode	# Recurrence Groups
Identical	44	No propagation	19
		Quick detection	19
		Further corruption	3
		Unable to classify	3
Different	19	No propagation	1
		Quick detection	6
		Further corruption	11
		Unable to classify	1
Unable to classify	6		

The above results indicate that an automatic diagnosis of recurrences based on symptoms may be possible. The following sections develop a system-independent approach for automatically diagnosing recurrences based on their symptoms.

7.2 Failure Diagnosis

We are looking at environments where many users run the same software, such as an operating system. Figure 7.4 shows a simplified picture of software development and service. Once software is developed and released to the field, many users run the software and report problems. There are usually several lines of service. Problems are diagnosed, fixes are made, and interim versions of software are released to the field. This process, represented by a loop at the lower right half of Figure 7.4, is called *software service*. Currently, problem diagnosis is performed manually. As a result, it can take a month before a problem report reaches the analyst who can diagnose the problem. In the meantime, a great deal of human effort can be wasted.

Failure diagnosis requires experience, a detailed knowledge of the system, and extensive reasoning. Figure 7.5 shows a simplified picture of failure diagnosis. A memory dump captures the processor state at the time of a failure. Given a failure report and an

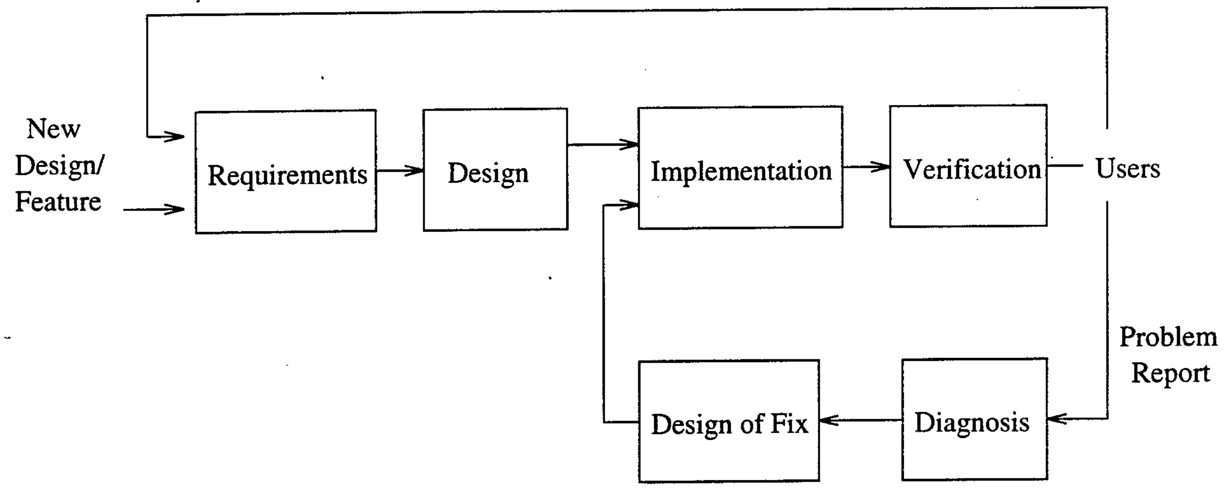


Figure 7.4 Software Development and Service

associated memory dump, analysts investigate key failure symptoms, such as the software function being executed, the apparent reason for the halt, and the error pattern. Based on these symptoms, they attempt to identify the underlying fault by reasoning back through the error generation and propagation processes. Software failure diagnosis is a complex task that is difficult to automate. Here we ask a simpler question: can we automate the diagnosis of recurrences? This question is significant, because the majority of field software failures are recurrences.

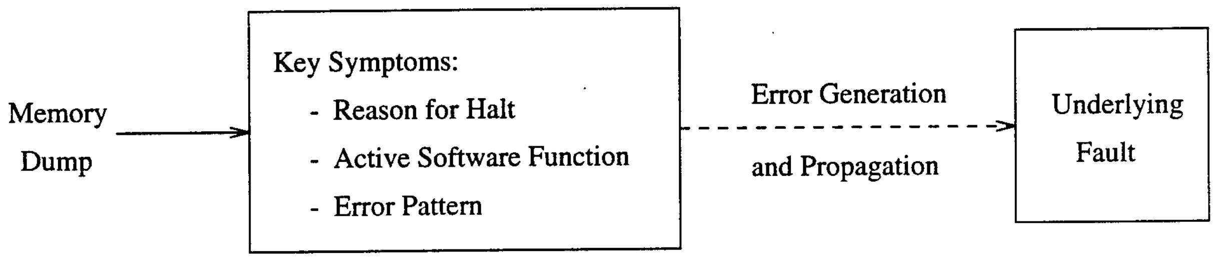


Figure 7.5 Software Failure Diagnosis

The results in the previous section showed that failures caused by the same software fault often share common symptoms, such as stack traces. This observation is not wholly new. Analysts have long used this knowledge as an aid for identifying recurrences, but they have done so manually. As a result, this knowledge has not been fully utilized. We focused on the feasibility of automating the diagnosis of recurrences based on symptoms, which naturally leads to the questions: what are the common symptoms; how to compare symptoms; and how effective will the diagnosis be. The following sections address these questions.

7.3 Diagnosis Strategy

A diagnosis strategy consists of a set of common symptoms and an associated matching scheme used for identifying recurrences. The diagnosis strategy is determined once by off-line evaluation. A diagnosis tool can be developed based on the selected diagnosis strategy.

7.3.1 Common symptoms

The first question is: which symptoms are usually shared by the failures caused by the same fault? Our analysis shows that failures caused by the same fault often share two types of symptoms: certain local and shared data (*data-oriented symptoms*), and characteristics of the code being executed at the time of failure (*code-oriented symptoms*).

Examples of code-oriented symptoms are stack traces (Figure 7.3 in Subsection 7.1.2) and code locations where problems were detected (also called *problem detection locations* or simply *detection locations*). In Figure 7.3, the first procedure from the top, except for the error-handling procedure, is called the procedure at the top of the stack (TOS). The procedure at the TOS and the associated offset (e.g., "%00365 into GET_ACCTENTRY" in Figure 7.3), when combined with the software version information, uniquely identifies the problem detection location. The software version must be known, because the procedure offset may change: there are many versions of the same software in the field because of bugfixes and functional enhancements. This issue is discussed further in Section 7.6. Examples of data-oriented symptoms are the values of parameters passed between procedures captured in a stack trace and the state of certain local and global variables.

Two extremes exist: a software fault can cause failures with different symptoms, and two software faults can cause failures with identical symptoms. Figure 7.6 illustrates the first extreme: two failures caused by the same software fault have different stack traces. In the figure, a circle represents a procedure call, and an arrow represents the execution within a procedure. Figure 7.6 shows a failure in which the base procedure MAINLOOP called the procedure NEXTREQ, which in turn called the procedure MONITORPRIMARY. MONITORPRIMARY called the procedure TK_PROCESS_TK_CKPT, in which a fault was exercised and a processor halt was asserted. In another failure, the same sequence was repeated, except that MAINLOOP reached MONITORPRIMARY through the procedure INITIALIZE. This calling path is also shown in the figure. Each chain of procedure calls forms a stack trace and is represented by a set of connected solid arrows in the figure. The dotted arrows represent a pair of procedure call and return that does not explicitly appear in a stack trace. Because the software structure is modular, there can be different program paths that reach the faulty code section. Figure 7.6 shows two such paths. Each of the paths gives a distinct stack trace.

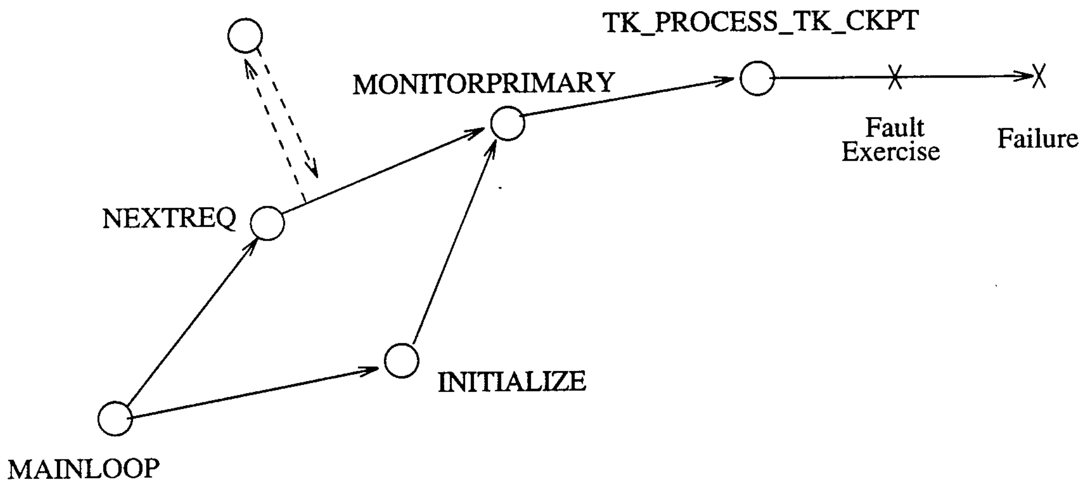


Figure 7.6 Detection near Faulty Code

Figure 7.7 shows another example of the first extreme. The figure shows a case in which a wide range of corruption occurred in shared data. The dotted lines represent accesses to the corrupt shared data. The underlying fault was a developer's misuse of a data structure. Once corruption of shared data occurs, any software function can detect some of the errors and assert a processor halt, which would lead to widely different stack

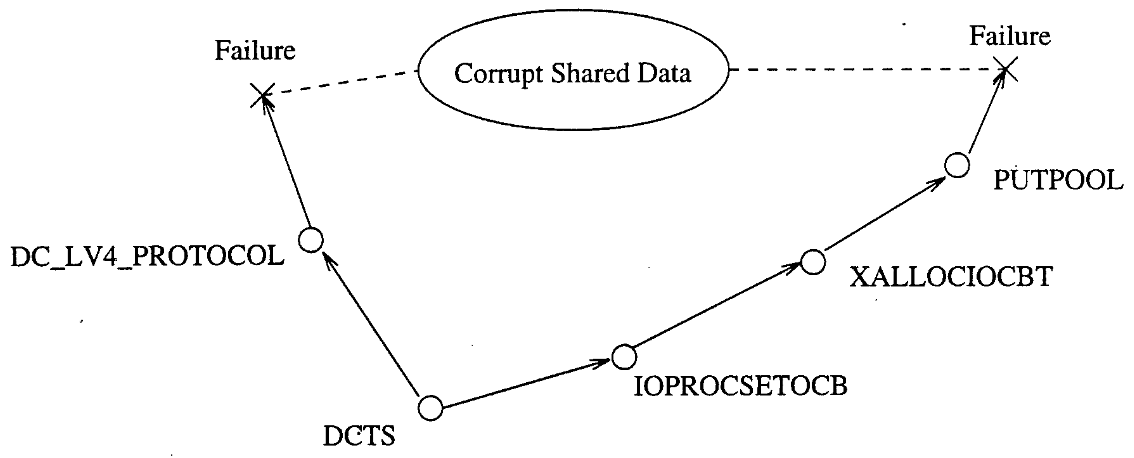


Figure 7.7 Detection after Corruption in Shared Data

traces, problem detection locations, and error patterns. Figure 7.7 shows two failures caused by the same software fault that have very different stack traces.

The second extreme to consider is that different faults can cause failures with identical symptoms. There are usually multiple consistency checks in a procedure, and each of the checks tests a different condition. As a result, problems caused by different software faults can be detected within the same procedure, and thus different software faults can cause failures with identical stack traces. In one case, a processor halt was asserted during the execution of the procedure DC_LV4_PROTOCOL, which was called by the base procedure DCTS. The underlying software fault was not providing a routine to handle a rare but legitimate sequence of events, which led to an inconsistent system state. This failure scenario and the left-hand-side stack trace in Figure 7.7 give an example of the second extreme.

7.3.2 Matching

Once a set of common symptoms is determined, the next question is: how do we compare failure symptoms (i.e., particular values of the common symptoms that were chosen to be used for the diagnosis)? Three types of matching can be considered: *complete matching*, *partial matching*, and *weighted matching*. In complete matching, two failures are declared to be caused by the same fault if their failure symptoms (e.g., two stack traces extracted from the two failures) are identical. In partial matching, two failures are declared to be caused by the same fault if their failure symptoms are within a certain distance of each other, based on a predefined measure of distance. Partial matching allows us to make a certain trade-off under the two extremes discussed in Subsection 7.3.1. Partial matching is discussed further in Section 7.5 and Section 7.6.

Weighted matching is necessary when several types of common symptoms are used. In weighted matching, a measure of the similarity of two failures is determined by comparing their values for each common symptom. These measures are then combined to form an overall measure that represents the similarity of two failures in their symptoms, based on the weight of each measure. The weights for different common symptoms can be determined by an iterative performance evaluation and based on the knowledge of software structure and functionality.

7.3.3 Proposed diagnosis strategy

We proposed the comparison of stack traces and problem detection locations as a strategy for identifying recurrences. Both complete and partial matching can be used for comparing stack traces. Since a detection location is a single piece of information, only complete matching can be used for matching detection locations. We focused on the use of code-oriented symptoms, because we used software failure reports (i.e., TPRs) generated by analysts, not actual dumps, in evaluating the effectiveness of the proposed diagnosis strategy. Full data-oriented symptoms exist in memory dumps, but they were not consistently recorded in the failure reports.

7.4 Diagnosis Environment

Figure 7.8 illustrates the type of automatic diagnosis environment envisioned. The diagnosis tool is connected with many user systems by an on-line alarm system. All previously reported failure symptoms and associated information, such as underlying faults and fixes, are stored in a database. On a failure alarm, the tool accesses the system that sent the alarm, extracts the values of the common symptoms (e.g., a stack trace and a detection location), and compares them with those of previously reported faults in the database. If a match is found in the database, the new failure is declared a recurrence of the corresponding fault; otherwise, it is declared a first occurrence. In the case of recurrence, the tool also identifies an available fix. After the diagnosis, the database is updated with new failure data. The environment shown in Figure 7.8 involves connections with many user systems and a database, as well as cooperation with other software service tools[1].

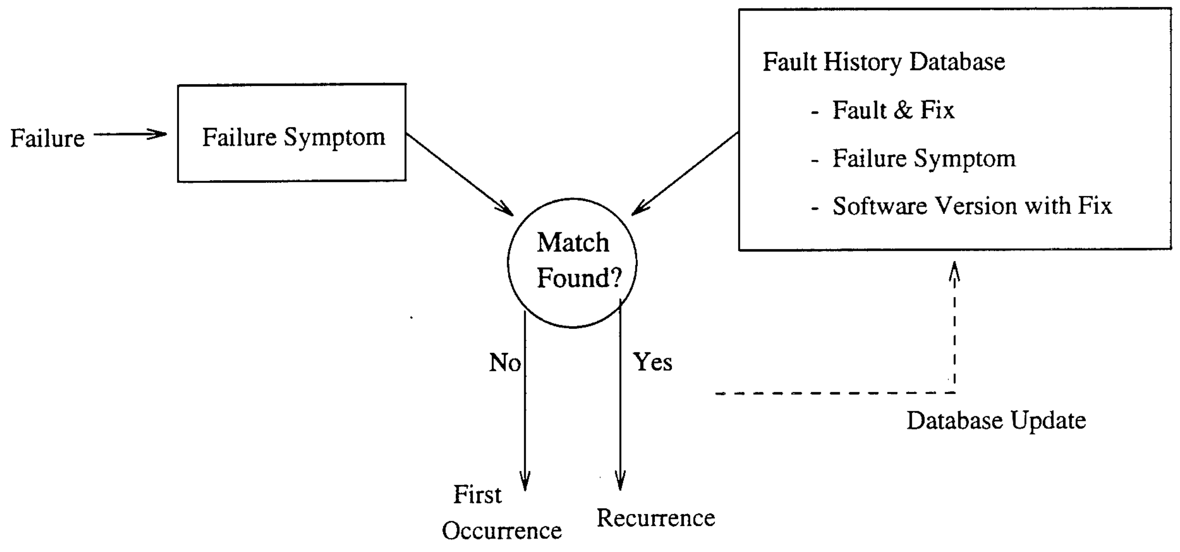


Figure 7.8 Diagnosis Environment Envisioned

7.4.1 Cost of misdiagnosis

A question here is: what is the cost of a misdiagnosis in an automated diagnosis environment? Suppose that two software faults (faults A and B) cause failures with identical symptoms. Let's assume that fault A causes a failure first. When fault A causes a failure for the first time, the failure will be diagnosed and fixed by analysts, because it is the first occurrence of fault A. When fault B causes a failure later at another site, an

[1]Such a fully integrated diagnosis environment is currently being built at Tandem Computers Incorporated.

automatic diagnosis tool will treat the failure as a recurrence of fault A and recommend the installation of a fix of fault A. Effectively, an incorrect fix is installed. The situation can be repeated when fault B causes failures at other sites. This problem can be resolved when fault B causes a failure at a site that is supposed to have a fix for the fault (i.e., a site that has installed a fix for fault A). Then the tool will realize the problem and draw human attention. To handle this situation, the failure database has to contain information about the software version that contains a fix for each fault in the database, as shown in Figure 7.8.

Thus, the cost of a misdiagnosis is the time between the initial incorrect diagnosis and the eventual correct diagnosis. Meanwhile, multiple failures can occur as a result of a single misdiagnosis. Considering the implementation of a diagnosis strategy as an automatic tool, more emphasis can be given to reducing the probability of misdiagnosis than to increasing the probability of successful diagnosis. We can also consider a semiautomated diagnosis environment in which a diagnosis tool provides the results of comparing failure symptoms and human analysts make the final decision for each case.

7.5 Evaluation Method

The effectiveness of a diagnosis strategy must be evaluated using actual data. This section discusses and applies a method for such evaluation. Note that this type of evaluation is performed only once as a means of determining the diagnosis strategy.

To evaluate the effectiveness of a diagnosis strategy under the two extremes described in Subsection 7.3.1, we considered *fault clusters* and *symptom clusters.* A fault cluster consists of all failures caused by a software fault. Fault clusters are formed based on diagnosis and repair logs by analysts. Given a set of failures for which the underlying software faults were identified, fault clusters can be uniquely determined. The set of fault clusters is the reference data. A symptom cluster consists of all failures sharing common symptoms. As far as the diagnosis is concerned, failures within the same symptom cluster are regarded as manifestations of the same software fault. Each choice of common symptoms and an associated matching scheme (i.e., each diagnosis strategy) may give a new set of symptom clusters. With these definitions, in the automated diagnosis environment shown in Figure 7.8, the terms *matching* and *clustering* can be used interchangeably. That is, "found a matching symptom in the database" and "clustered with a symptom in the database" have the same meaning.

A one-to-one correspondence between fault clusters and symptom clusters would be ideal, but that correspondence is difficult to achieve. We considered two general situations to describe the imperfection of a diagnosis strategy: *join* and *split* (Figure 7.9). A join refers to a situation in which failures caused by different software faults are grouped into a single symptom cluster. A split refers to a situation in which failures caused by a single software fault are divided into multiple symptom clusters.

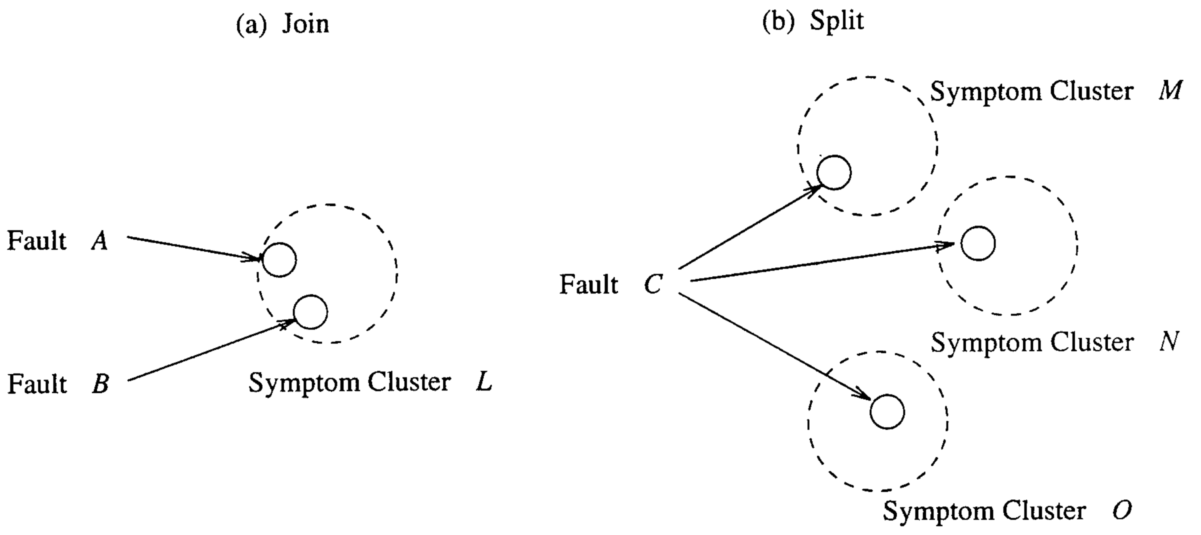


Figure 7.9 Join and Split

Figure 7.10 illustrates a join and a split when the complete matching of stack traces is used. In the first group of failures caused by fault A, the underlying fault was not providing a routine to handle a legitimate operational scenario when designing the software. When fault A was exercised, a pointer in a data structure acquired an invalid address, because it was not updated (i.e., the data structure became inconsistent). The problem was detected by a consistency check on the first access of the data structure in all four failures. This situation was a typical example of no propagation discussed in Subsection 3.2.2. All failures caused by fault A had identical stack traces.

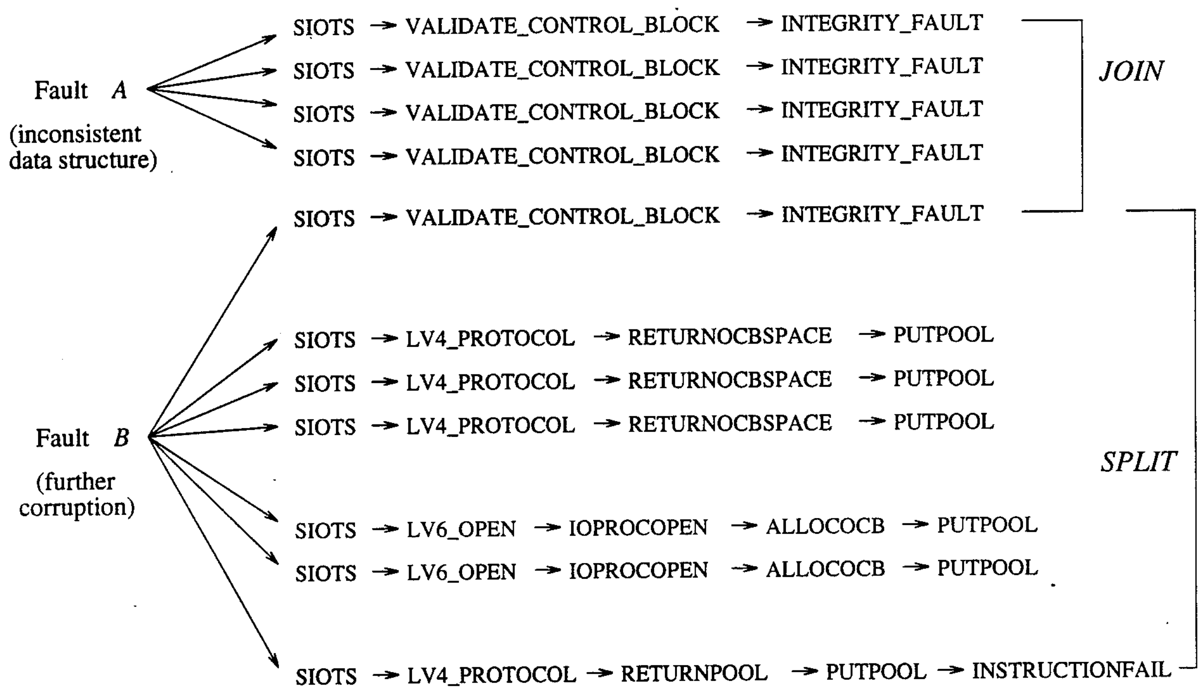

Fault A
(inconsistent data structure)
SIOTS → VALIDATE_CONTROL_BLOCK → INTEGRITY_FAULT
SIOTS → VALIDATE_CONTROL_BLOCK → INTEGRITY_FAULT
SIOTS → VALIDATE_CONTROL_BLOCK → INTEGRITY_FAULT
SIOTS → VALIDATE_CONTROL_BLOCK → INTEGRITY_FAULT
SIOTS → VALIDATE_CONTROL_BLOCK → INTEGRITY_FAULT
JOIN
Fault B
(further corruption)
SIOTS → LV4_PROTOCOL → RETURNOCBSPACE → PUTPOOL
SIOTS → LV4_PROTOCOL → RETURNOCBSPACE → PUTPOOL
SIOTS → LV4_PROTOCOL → RETURNOCBSPACE → PUTPOOL
SPLIT
SIOTS → LV6_OPEN → IOPROCOPEN → ALLOCOCB → PUTPOOL
SIOTS → LV6_OPEN → IOPROCOPEN → ALLOCOCB → PUTPOOL
SIOTS → LV4_PROTOCOL → RETURNPOOL → PUTPOOL → INSTRUCTIONFAIL

Figure 7.10 Examples of Join and Split

In the second group of failures caused by fault B in Figure 7.10, the underlying fault was a misuse of a data structure, which resulted in a wide range of corruption of shared data. This situation was a typical example of further corruption discussed in Subsection 3.2.2. Depending on which of the generated errors are detected first, the failures caused by fault B can have very different stack traces. The figure shows four distinct stack traces.

The data structure affected when fault A was exercised was a part the memory area that was corrupted when fault B was exercised. As a result, one of the seven failures caused by fault B and the failures caused by fault A had identical stack traces, which is a join. The effect of a join is the potential for misdiagnosis. An automatic diagnosis tool may not be able to distinguish the faults that cause a join. Two situations are possible. First, a failure caused by a new fault can be declared a recurrence of a previously reported fault. Second, a recurrence of a fault can be declared a recurrence of another fault.

The failures caused by fault B in Figure 7.10 form four symptom clusters, which is a split. The effect of a split is the potential for repeated diagnoses of the same fault. Even with an automatic diagnosis tool, fault B has to be diagnosed four times by analysts. The current manual diagnosis strategy would require that fault B be diagnosed seven times by analysts. A perfect, automatic diagnosis tool would require that fault B be diagnosed only once by analysts.

Suppose that using a particular diagnosis strategy leads to N joins and M splits. Also, let J_i denote the number of faults involved in the i-th join, and let S_j denote the number of symptom clusters involved in the j-th split. Then, the following measures of effectiveness can be defined:

$$\begin{aligned} F_{fault-misdiagnosed,max} &\equiv \text{Maximum number of faults misdiagnosed} \\ &= \sum_{i=1}^{N} (J_i - 1) , \qquad (7.1) \end{aligned}$$

$$\begin{aligned} F_{repeated-diagnosis,max} &\equiv \text{Maximum number of repeated diagnoses} \\ &= \sum_{i=1}^{M} (S_i - 1) , \qquad (7.2) \end{aligned}$$

$$\begin{aligned} S_{correct-diagnosis,min} &\equiv \text{Minimum number of recurrences diagnosed correctly} \\ &= \text{Total number of recurrences} - F_{repeated-diagnosis,max} . \qquad (7.3) \end{aligned}$$

The actual number of faults misdiagnosed can be smaller than $F_{fault-misdiagnosed,max}$ for the following reasons:

- Overlaps of joins: for example, two software faults can generate two symptom clusters as a result of two joins and two splits, as shown in Figure 7.11. In this case, the actual number of faults misdiagnosed is usually one, not two, as calculated from Equation (7.1).
- Nonoverlap of fault manifestation windows: multiple software faults cause failures with identical symptoms in disjoint time windows. Figure 7.12 illustrates a case

involving two faults, in which one fault causes a failure after the other is completely fixed in the field. This nonoverlap usually happens when the second fault, which has been dormant, becomes active after a code update. In the actual evaluation performed in the next section, if two fault manifestation windows are more than six months apart, it is assumed that the join does not cause a misdiagnosis.

The measures $F_{repeated-diagnosis,max}$ and $S_{correct-diagnosis,min}$ provide a maximum and a minimum number, respectively, because there can be overlaps in splits. For example, suppose that the actual sequence of failure occurrences over time in Figure 7.11 is $\{p, s, q, r\}$ and that there is enough time for diagnosis between any two consecutive failures. Then the actual number of repeated diagnoses in this case can be one, not two, as calculated from Equation (7.2), because the diagnosis tool can take advantage of information that faults A and B can cause failures with identical symptoms after the occurrence of failure q. When failure r occurs, the tool can recommend the installation of fixes for both faults.

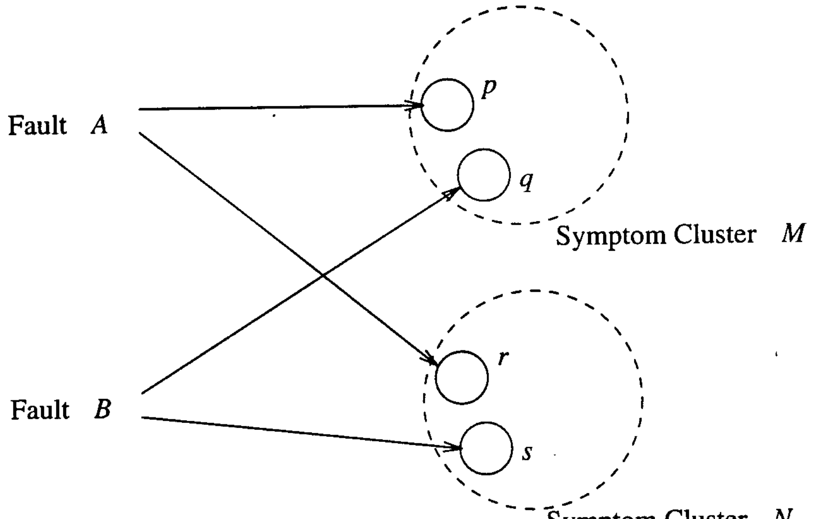


Figure 7.11 Overlap of Joins and Splits

Partial matching uses a less strict rule than complete matching in building symptom clusters and therefore generates fewer symptom clusters. As a result, when compared with complete matching, partial matching leads to a greater or equal number of joins

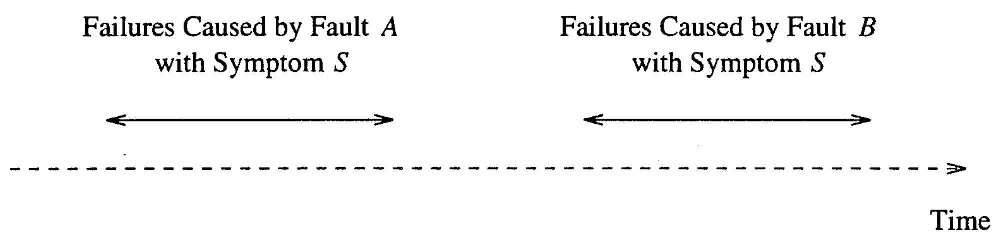

Failures Caused by Fault *A* with Symptom *S*
Failures Caused by Fault *B* with Symptom *S*
Time

Figure 7.12 Nonoverlap of Fault Manifestation Windows

and to a lesser or equal number of splits. Therefore, partial matching can be used to reduce the probability of repeated diagnosis at the cost of increasing the probability of misdiagnosis.

7.6 Evaluation of the Proposed Diagnosis Strategy Using Field Data

This section evaluates the effectiveness of the proposed diagnosis strategy using the field failure and repair data from two Tandem system software products. The intent of the evaluation is to investigate the range of effectiveness of the proposed diagnosis strategy and its variations. Ideally, we would have evaluated the strategies using all failures. Because of time constraints, we used data from two Tandem system software products. Given this limit, we selected two products that have widely different reputations among Tandem analysts in terms of quality, hoping that an evaluation using failures in the two products would give us a range of effectiveness.

One product implements the low-level functions to support database applications and is referred to as DB. The other product implements network communication functions and is referred to as DC. These products run as processes and serve requests from user applications. Among analysts, DB is known to be robust, while DC is known to be not robust.

We first extracted all user-generated TPRs caused by faults in the two system software products for the past two to three years. We then extracted all preceding TPRs caused by the same faults. During the measurement period, the products were modified many times to fixes software faults and to enhance their features. There was also a major revision. Both products are written in Transaction Application Language (TAL), which is similar to C. The size of each product is on the order of 10^5 lines of commented source code.

7.6.1 Evaluation using failures in DB

Table 7.2 shows a breakdown of the 152 software failures caused by faults in DB, based on how the problems were detected. The numbers inside parentheses further subdivide a class. The failures were caused by 55 faults. The table shows that about 85% percent of the problems were detected during the execution of the DB code, and 72% of the problems were detected by the consistency checks in DB. The percentages are lower in DC (64% and 33% respectively in failures caused by faults in DC), which indicates that the DB software has better error detection and error containment capabilities. This finding corroborates the reputations of the two products among analysts. Only the 130 failures detected during the execution of the DB code were used for the analysis, because these failures and the failures detected outside DB naturally have different code-oriented symptoms.

Table 7.2 Problem Detection Profile (DB)

Problem Detection	Fraction (%)
Consistency Checks	81
Detection within DB	(72)
Detection outside DB	(9)
Virtual Memory Protection	14
Detection within DB	(13)
Detection outside DB	(1)
Hang	5

7.6.1(a) Matching stack traces

Table 7.3 shows the effectiveness of the diagnosis when symptom clusters were built by the complete matching of stack traces. Although the stack trace exists in all failures, not all TPRs contained stack traces. This omission usually happened when there were many recurrences of a software fault. In TPRs reporting later occurrences, analysts sometimes left pointers to the TPRs that analyzed earlier occurrences, rather than detailing symptoms. Our experience shows that this omission usually happens when later occurrences share the same symptoms with earlier occurrences. Out of 130 TPRs, 78 contained stack traces. These failures were caused by 39 unique faults. Note that the recurrence rate in the data set became much lower than its actual value. The average number of procedures in a stack trace (i.e., the average length of a stack trace) was 5.7.

Table 7.3 Complete Matching of Stack Traces (DB)

Common Symptom	#Joins	#Splits	$F_{fault-misdiagnosed,max}$	$S_{correct-diagnosis,min}$
Stack Trace (78 TPRs, 39 faults)	2	9	2{0}	28

{}: nonoverlap of fault manifestation windows

Table 7.3 shows that, with the complete matching of stack traces, at least 72% (28 out of 39) of the recurrences could have been identified correctly. (We think that this percentage would be higher if all TPRs contained stack traces.) At most, two faults could have been misdiagnosed. In each of the two joins, two software faults affected the processor state in the same manner, and a table entry was lost when the faults were exercised. The problems were detected during an attempt to locate a nonexisting entry. The problems were detected at an identical code location during the execution of the same software function. The data showed that, in each join, the two software faults that caused the joins had nonoverlapping manifestation windows. Therefore, the actual number of faults misdiagnosed was zero, which is shown inside a pair of braces in Table 7.3. Including the halt code in building symptom clusters had a negligible effect (it reduced $S_{correct-diagnosis,max}$ by one), because many failures were detected by consistency checks and all of them had identical halt codes.

Partial matching can reduce the number of splits, but this reduction is accomplished at the cost of increasing the number of joins. We investigated the patterns of stack traces in the nine splits in Table 7.3. Two major patterns of differences in the splits were: the stack traces are very different from one another, and the stack traces are the same except for minor differences in the middle. The first pattern is difficult to resolve with partial matching. The second pattern occurred primarily due to the existence of different program paths to reach the same errors. As a result, different stack traces causing these splits often had identical procedures at the TOS. Based on these observations, the following heuristics were considered for the partial matching of stack traces:

(1) If two stack traces have the same length and differ from one another by no more than one procedure, group them into the same symptom cluster. This heuristic is called *differ-by-one*. Note that repeated applications of this heuristic can cluster stack traces that differ by more than one procedure.

(2) Apply the differ-by-one heuristic only if the procedures at the TOS are the same.

(3) If one stack trace contains all of the procedures in the other without regard to their order, group them into the same symptom cluster. This heuristic is called *contain-the-other*.

(4) Apply the contain-the-other heuristic only if the procedures at the TOS are the same.

Table 7.4 shows the results of the partial matching of stack traces. The numbers inside the parentheses indicate the differences from the numbers obtained when complete matching is used (Table 7.3). The table shows that the procedure at the TOS is a useful common symptom. Including it prevented the increase in the number of joins appreciably. By applying the differ-by-one heuristic only when the procedures at the TOS are identical, at least 87% (34 out of 39) of the recurrences could have been identified correctly. The number of joins increased by one, but the actual number of faults misdiagnosed was still zero because of the nonoverlap of fault manifestation windows. The contain-the-other heuristic was not effective.

Table 7.4 Partial Matching of Stack Traces (DB)

Heuristics	#Joins	#Splits
Differ-by-one	(+5)	(-4)
Differ-by-one & the same procedure at the TOS	(+1)	(-4)
Contain-the-other	(+8)	(0)
Contain-the-other & the same procedure at the TOS	(+1)	(0)

7.6.1(b) Matching problem detection locations

The results in Table 7.4 indicated that the problem detection location can be a useful common symptom. The information identifying a detection location consists of a procedure name and associated offset (Figure 7.3). However, the offset part of the symptom (e.g., %00365 in "%00365 into GET_ACCTENTRY" shown in Figure 7.3) is version sensitive. Note that there are many versions of the same software in the field because of bugfixes and functional enhancements.

DB developers designed the software such that, when nonrecoverable errors are detected by consistency checks, an ASCII string (called the *symptom string*) is inserted at the designated location of the process stack before a processor halt is asserted. Given a symptom string, analysts can recognize the detection location regardless of the software version. A symptom string consists of three parts that identify, respectively, the source file name, the procedure name, and the specific software check that detected a problem. All 110 TPRs reporting failures detected by the DB consistency checks (Table 7.2) contained symptom strings. Therefore, the effectiveness of the diagnosis with the comparison of detection locations was evaluated based on the 110 TPRs. These TPRs were caused by 39 software faults.

Table 7.5 shows the effectiveness of the diagnosis when symptom clusters were formed based on symptom strings. Since a symptom string is a single piece of information, only complete matching is possible. Table 7.5 shows that at least 94% (67 out of 71) of the recurrences could have been identified correctly by matching symptom strings. At most, eight faults could have been misdiagnosed. The data showed that the actual maximum number of faults misdiagnosed was one because of the nonoverlap of fault manifestation windows.

Table 7.5 Matching Detection Locations Using Symptom Strings (DB)

Common Symptom	#Joins	#Splits	$F_{fault-misdiagnosed,max}$	$S_{correct-diagnosis,min}$
Symptom string (110 TPRs, 39 faults)	5	4	8{1}	67

{}: nonoverlap of fault manifestation windows

Whether a diagnosis strategy is better than another can be determined by a hypothesis test. This issue was treated lightly because we used failures in only two products. The hypothesis that matching symptom strings and the complete matching of stack traces were equally effective in terms of successful diagnosis was rejected, which indicates that matching symptom strings was more effective in terms of successful diagnosis for the measured period in DB (Table 7.3 and Table 7.5). We tested the hypothesis using the binomial test at the 5% significance level, treating the diagnosis of recurrences as Bernoulli trials. The hypothesis that matching symptom strings and the complete matching of stack traces were equally effective in terms of misdiagnosis was not rejected by the same test at the same level. One caution regarding the observations is that the two tables used for the comparison were generated using data sets with different recurrence rates, because analysts did not always record stack traces in TPRs.

A limitation of using symptom strings for comparing problem detection locations is that symptom strings exist only when problems are detected by consistency checks. (This issue is discussed further in Subsection 7.6.2.) Note that a stack trace always exists, even in failures caused by nonsoftware faults.

We also used two variations of the symptom string to form symptom clusters: the procedure at the TOS, and the procedure at the TOS and associated offset. These symptoms always exist. Table 7.6 shows the effectiveness of the diagnosis when these symptoms are used. Although the three sets of TPRs used to generate Tables 7.5 and 7.6 were different, several observations can be made. Compared with the use of the symptom string, use of the procedure at the TOS increased $F_{fault-misdiagnosed,max}$, because some problems caused by different faults can be detected at different locations within the same procedure. Use of the procedure at the TOS and an associated offset increased the number of splits appreciably, because the same code location can have different offset values in different software versions. One interesting observation is that the number of joins decreased because of the nonoverlap of fault manifestation windows between different software faults causing a join. Because of code changes between the windows, the failures caused by the faults had different offsets, although they were detected at an identical location.

Table 7.6 Matching Detection Locations Using Variations of the Symptom String (DB)

Common Symptom	#Joins	#Splits	$F_{fault-misdiagnosed,max}$	$S_{correct-diagnosis,min}$
Procedure at the TOS (130 TPRs, 43 faults)	5	5	13	82
Procedure at the TOS & offset (110 TPRs, 40 faults)	3	11	4	54

7.6.2 Evaluation using failures in DC

Table 7.7 shows a breakdown of the 258 software failures caused by 72 faults in DC. The evaluation was performed based on 166 failures that were detected during the execution of the DC code and contained stack traces. These failures were caused by 59 software faults. The average number of procedures in a stack trace was 3.6.

Table 7.7 Problem Detection Profile (DC)

Problem Detection	Fraction (%)
Consistency Checks	51
Detection within DC	(33)
Detection outside DC	(19)
Virtual Memory Protection	46
Detection within DC	(31)
Detection outside DC	(15)
Hang	3

7.6.2(a) Matching stack traces

Table 7.8 shows the effectiveness of the diagnosis when the complete matching of stack traces was used. Using halt codes together with stack traces reduced the number of joins while not increasing the number of splits, because the percentage of the problems detected by consistency checks was lower. Therefore, the halt code, which represents how problems were detected, became a useful common symptom. In the subsequent analysis, failures with different halt codes were not grouped into the same symptom cluster.

Table 7.8 Complete Matching of Stack Traces (DC—166 TPRs Caused by 59 Faults)

Common Symptom	#Joins	#Splits	$F_{fault-misdiagnosed,max}$	$S_{correct-diagnosis,min}$
Stack Trace	13	11	21	77
Stack Trace & halt code	10	11	16{6}	77

{}:nonoverlap of fault manifestation windows and using subprocedure traces

In four of the ten joins in Table 7.8, problems caused by different software faults were detected at identical code locations during the execution of the same software function. In the remaining six joins, problems were detected at different locations within the same procedure. These joins were primarily due to the existence of big procedures that detected errors caused by different faults. The existence of big procedures is attributed to the language's support of subprocedures, callable only within a procedure. The data showed that, when stack traces and subprocedure traces within the procedures at the TOS are used together, $F_{fault-misdiagnosed,max}$ is reduced to eight, without $S_{correct-diagnosis,min}$ being affected. This result indicates that the effectiveness of the diagnosis when matching stack traces can be improved by proper sizing of procedures.

The maximum number of faults misdiagnosed was again reduced to six because of the nonoverlap of fault manifestation windows. With the complete matching of stack traces, halt codes, and subprocedure traces within the procedure at the TOS, at least 72% (77 out of 107) of the recurrences could have been identified correctly. At most, six faults could have been misdiagnosed. There was no significant difference in the performance of the complete matching of stack traces in the two software products in terms of successful diagnosis, but the complete matching of stack traces was more effective in DB than in DC in terms of misdiagnosis (Table 7.3 and the second row of Table 7.8). These observations were confirmed by the binomial test at the 5% significance level. Again, a caution is that the recurrence rate in the data set used for DB was lower than its actual value.

Table 7.9 shows the results of a classification of the 11 splits in Table 7.8, based on the major reasons for the splits. "Data corruption" refers to cases in which a software fault caused corruption in a shared data area. If such corruption occurs, errors can be detected during the execution of different software functions, which is why a software fault causes failures with different stack traces. There were two subtle software faults (i.e., two splits) causing corruption in shared data. It took a long time to diagnose the problems, and meanwhile, the faults caused failures with 23 different stack traces. That is, the two faults accounted for 21 of the 23 potential repeated diagnoses.

Table 7.9 Breakdown of Splits (DC)

Reason for Split	#Splits	$F_{repeated-diagnosis,max}$
Data corruption	4	23
Different calling sequence	6	6
Data dependence	1	1

"Different calling sequence" refers to cases in which differences in stack traces are attributed to different program paths to reach and detect the same errors. "Data dependence" refers to cases in which, depending on the actual values of errors and the machine state, a problem is detected at different (but typically nearby) code locations. In the actual case, the difference in stack traces was one extra procedure at the TOS. This type of difference in stack traces could also be observed in some data corruption cases. For example, when a software function accesses a corrupt data region, depending on the actual values of errors and the machine state, a problem could be detected after an additional procedure call, after a return to the previous procedure, or at different locations within that procedure. With this observation, we added the fifth heuristic for the partial matching of stack traces (Table 7.4):

(5) Given two stack traces, if one is longer than the other by one procedure and the difference is an additional procedure at the TOS, group them into the same symptom cluster. This heuristic is called *extra-procedure-at-TOS*.

Table 7.10 shows the effectiveness of the diagnosis when the partial matching of stack traces was used. The numbers inside the parentheses indicate the differences from the numbers obtained when complete matching is used (the second row of Table 7.8). Subprocedure traces were not used here. To avoid an excessive increase in the number of joins, the differ-by-one heuristic was not applied to the stack traces that contain only one procedure, and the contain-the-other heuristic was not applied to the stack traces that contain one or two procedures. All heuristics increased $S_{correct-diagnosis,min}$, but not drastically, indicating that the partial matching hueristics could not completely capture the randomness in failure symptoms caused by corruption in a shared data area. This result indicates that the error containment capability of software can be a major factor that affects the effectiveness of the diagnosis. The increases in $F_{fault-misdiagnosed,max}$ were primarily because of short stack traces (containing three or fewer procedures) that easily caused joins when partial matching was used. Table 7.10 shows that the procedure at the TOS helped to suppress the increase in the number of joins in DC, also.

Table 7.10 Partial Matching of Stack Traces (DC)

Heuristic	#Joins	#Splits	$F_{fault-misdiagnosed,max}$	$S_{correct-diagnosis,min}$
Differ-by-one	(+4)	(0)	(+12)	(+7)
Differ-by-one & the same procedure at the TOS	(+1)	(0)	(+3)	(+6)
Contain-the-other	(+2)	(-1)	(+5)	(+6)
Contain-the-other & the same procedure at the TOS	(0)	(0)	(+3)	(+4)
Extra-procedure-at-TOS	(+3)	(-2)	(+3)	(+2)

7.6.2(b) Matching problem detection locations

The product DC was not designed to provide the symptom string. Although not all TPRs recorded the failed software version, it was possible to determine whether two problems were detected at the same code location, based on the information in TPRs (stack traces, offsets, halt codes, and textual descriptions by analysts) and the actual code. Therefore, in the following evaluation, it was assumed that DB-style symptom strings existed in all failures. That is, we assumed that an automatic diagnosis tool can compare the detection locations in two failures regardless of the software version. Symptom clusters were formed based on the following three symptoms, listed in the increasing order of strictness:

(1) Procedure at the TOS

(2) Symptom string

(3) Symptom string and stack trace

Table 7.11 shows that, by matching symptom strings and halt codes, at least 78% (83 out of 107) of the recurrences could have been identified correctly. At most, six faults could have been misdiagnosed. For the measured period, there was no significant difference between the complete matching of stack traces and the matching of symptom strings in their performance in DC (the second rows of Table 7.8 and Table 7.11). A comparison of Table 7.5 and the second row of Table 7.11 shows that the use of symptom strings was more effective in DB than in DC in terms of successful diagnosis, but the use of symptom strings showed similar performance in the two products in terms of misdiagnosis. These observations were again confirmed by the binomial test at the 5% significance level.

Table 7.11 Matching Detection Locations (DC—166 TPRs Caused by 59 Faults)

Common Symptom	#Joins	#Splits	$F_{fault-misdiagnosed,max}$	$S_{correct-diagnosis,min}$
Procedure at the TOS	15	10	25	89
Symptom string	8	12	11{6}	83
Symptom string & stack trace	6	14	6	70

{}: nonoverlap of fault manifestation windows

7.6.2(c) Version independent problem detection location

Now a question is: how does an automatic tool compare the two detection locations in DC? The implementation of the DB-style symptom string can be encouraged in all products. But the percentage of failures that have the symptom string (i.e., the percentage of the failures that are detected by consistency checks) will depend on software quality. Besides, the value of the percentage can be estimated after the software is released to the field.

Here we propose two approaches. First, the diagnosis tool can maintain differences in offsets among different software versions. While this approach guarantees a comparison of two detection locations, it requires additional work of cross-referencing software versions in generating a new version, and the size of the failure database may grow rapidly. Second, the *machine-code symptom string* can be used. The machine-code symptom string is defined as machine instructions in the binary form, before and after a detection location. Just as a stack trace, the machine-code symptom string always exists. (There may be rare cases in which we cannot compare machine-code symptom strings if two detection locations are at different edges of two memory pages and the connecting pages are not available.) A possible strategy is to use the DB-style symptom string, if available, or to use the machine-code symptom string.

7.7 Summary

This chapter presents a system-independent approach for automatically diagnosing recurrences based on symptoms, for use in environments where many users run the same software. The approach is based on observations that the majority of field software failures in such environments are recurrences and that failures caused by the same software fault often share common symptoms. Specifically, we proposed the comparison of stack traces and problem detection locations as a strategy for identifying recurrences. We applied this strategy using failures in two Tandem system software products and compared the results obtained with actual diagnosis and repair logs from analysts.

The comparison showed that between 75% and 95% of recurrences can be successfully identified by matching failure symptoms, such as stack traces and detection locations. Less than 10% of faults are misdiagnosed. The results show that the proposed automatic diagnosis of recurrences allows analysts to diagnose only one out of several software failures (i.e., primarily the failures caused by new faults). In the case of a recurrence for which the underlying cause was identified, the diagnosis tool can rapidly provide a solution. In the case of a recurrence for which the underlying cause is being investigated, the diagnosis tool can prevent a repeated diagnosis by identifying previous failures caused by the same fault. These benefits are not free of cost. Misdiagnosis is harmful, because a single misdiagnosis can result in multiple additional failures. (Such a danger exists in diagnoses by analysts, also.) The proposed approach needs to be implemented in a pilot. Measurements need to be made to determine how well the approach works in real environments and to make design trade-offs.

The results also indicated that the error containment capability of the software can be a major factor determining the effectiveness of the diagnosis. Proper sizing of procedures can also be a factor when stack traces are used.

The numerical results reported here are specific to the measurements. However, the two measured products used for the evaluation consist of many small procedures and are written in a high-level language, characteristics common in system software products. Our experience shows that there are no special requirements the software must satisfy for the approach to be effective. Still, further experiments are necessary to determine how well the numbers will project to other system software products.

There are several areas that require more research. First, more diagnosis strategies have to be investigated. For example, the use of data-oriented symptoms and associated clustering strategies have to be investigated based on actual dumps. Second, it is necessary to use failures from more software products for evaluations, because in real environments, many products run together and the effects of faults can cross the boundaries between the products. Failures caused by nonsoftware faults also have to be included in the evaluation, because determining whether a failure was caused by a software fault is not always straightforward. Finally, it would be interesting to investigate the effectiveness of the approach for application software products and for diagnosing problems less severe than operating system crashes.

Chapter 8

Conclusions

The contribution of this thesis is in identifying and addressing critical dependability issues for large, continually evolving, operational software. Using measurements collected from the Tandem GUARDIAN operating system, this thesis demonstrates how to develop analysis techniques for evaluating the dependability of operational software while taking design issues into account. This research brings practical issues in designing and maintaining large software systems together with theoretical issues such as problem diagnosis, fault tolerance, and modeling and analysis. This thesis consists of two major parts: analysis and design. The analysis covered software fault categorization and characterization of software error propagation, identification of software fault tolerance of process pairs, evaluation of the impact of software faults on the overall system, and the development of techniques for analyzing multiway failure dependencies among software and hardware modules. Based on the results of analysis, this thesis developed a system-independent approach for automatically diagnosing recurrences based on symptoms.

The next three sections summarize the measurements and the major results of the analysis and design from this research. The numerical results are specific to measurements, but the methods and principles apply to other studies. The thesis concludes with a section that discusses possible extensions of this work.

8.1 Measurements

Measurements were taken from a fault-tolerant software system: the Tandem GUARDIAN operating system. Two types of data were used: human-generated software failure reports and on-line processor halt logs automatically generated by the operating system. Human-generated software failure reports contain detailed information about the underlying faults, failure symptoms, and fixes. Only the reports generated as a result of field software failures (i.e., software failures which occurred in user systems) were used. On-line processor halt logs provide close to 100% of reporting and accurate timing information on processor failure and recovery. Processor halt logs taken from two in-house Tandem systems (a Tandem Cyclone and a Tandem VLX systems) were used.

Software failure reports were used for categorizing the underlying faults, for investigating failure symptoms, for evaluating the software fault tolerance of process pairs, for developing a reliability model for operational software, and for developing a symptom-based diagnosis strategy for automatically diagnosing recurrences. Processor halt logs were used for developing a method for analyzing multiway failure dependencies and for evaluating the improvement in service achieved by the single-failure tolerance of the measured system.

8.2 Analysis

8.2.1 Fault categorization and characterization of error propagation

We explored new ground for building software fault models from the software fault tolerance perspective. In addition to categorizing the underlying faults of software failures, we identified the immediate effects of the faults on the processor state and traced the propagation of the effects on other system areas until problems were detected by the operating system.

The results showed that about 72% of reported field software failures in Tandem systems are recurrences of previously reported faults. This result shows that, in environments where many users run the same software, the number of faults in software is not the only important factor. Recurrences can seriously degrade software dependability in the field.

Missing operations and not providing routines to handle rare but legitimate operational scenarios are the most common types of software faults in Tandem systems. The data showed that there is a 60% chance that a single program variable acquires an initial, incorrect value when software faults are exercised. In about 20% of the cases, multiple program variables are affected simultaneously. Once errors are generated, the three major error propagation modes are: the first error is certain to be detected on the first access by consistency checks (no propagation, 31%); the problem is detected shortly after the first error is accessed and used (quick detection, 39%); and the first error causes more errors, which are detected after a significant latency (further corruption, 18%). In about half of the failures, problems are detected by consistency checks; in the other half, problems are detected as a result of address violations.

The investigation of failure symptoms showed that failures caused by the same software fault often share identical stack traces, which suggests that automatic diagnosis of recurrences based on symptoms might be possible. Further analysis showed that error propagation (further corruption) and modular program structure are major reasons that failures caused by the same software fault have different stack traces. No propagation and quick detection are major reasons that failures caused by the same software fault have identical stack traces. Consistency checks help failures caused by the same software fault have identical stack traces by preventing error propagation. These results formed the basis for our development of an approach to automatically diagnose recurrences based on symptoms.

8.2.2 Evaluation of software fault tolerance

Software fault tolerance of process pairs in the Tandem GUARDIAN system was evaluated using two types of measurements: human-generated software failure reports and on-line processor halt logs automatically generated by the operating system.

The results showed that hardware fault tolerance buys software fault tolerance. Using process pairs in Tandem systems allows the system to tolerate about 70% of reported faults in the system software that cause processor failures. This result shows that, in a distributed transaction-processing environment, a significant level of software fault tolerance can be achieved by the use of checkpointing and restart, a technique for tolerating hardware faults. The loose coupling between processors, which results in the backup execution (the processor state and the sequence of events occurring) being different from the original execution, is a major reason for the measured software fault tolerance.

The results indicated that the actual level of software fault tolerance achieved by the use of checkpointing and restart depends on the degree of difference in the processing environment between the original execution and restart and on the proportion of subtle faults in the software. While process pairs may not provide perfect software fault tolerance, the implementation of process pairs is not as prohibitively expensive as is developing and maintaining multiple versions of large software programs.

The results of Markov reward analysis using processor halt logs showed that the single-failure tolerance of the measured system reduces the service loss incurred by software failures by 89%. This result corroborates the results obtained using software failure reports. The results also showed that the single-failure tolerance reduces the service loss incurred by nonsoftware failures by 92% and that software failures account for 30% of the service loss in the measured system.

8.2.3 Analysis of failure dependency

Failure dependency is a serious concern in parallel and fault-tolerant systems. Failure dependencies between two components can be analyzed using correlation coefficients.

However, such dependencies can exist among multiple components. We developed a method for analyzing multiway failure dependencies among software and hardware modules. The method is based on multivariate statistical techniques, such as factor analysis and cluster analysis. An illustration of the method using processor halt logs demonstrated that factor analysis can unearth the underlying multiway failure dependencies and that cluster analysis can identify the actual dependency patterns.

8.2.4 Reliability modeling of operational software

To identify the factors determining the dependability of operational software, we built a software reliability model that describes the impact of software faults on an overall Tandem system in the field, and we conducted reliability sensitivity analysis using the model. The results showed that, in addition to the conventional approach of reducing the number of faults in software, software dependability in Tandem systems can be enhanced by reducing the recurrence rate and by improving the robustness of process pairs and the system configuration. The number of faults in software and the recurrence rate are general factors; the robustness of process pairs and the system configuration are platform-dependent factors. Improvement efforts can be optimized by estimating the cost of improving each factor.

The results also showed that the impact of software fault tolerance (i.e., the robustness of process pairs) and the impact of system configuration are as significant as is the impact of the number of faults in software. A complete elimination of recurrences in the measured system would triple the software mean time between failures. The analysis of reliability sensitivity to fault category showed that faults such as missing operations and not providing routines to handle rare but legitimate operational scenarios are the major causes of software reliability loss. Additional code inspection and testing efforts can be directed to these types of faults. The investigation of the impact of system configuration on software dependability demonstrated the importance of considering interactions between software and hardware in the context of an overall system.

8.3 Design

After all analyses were completed, the issue of recurrence stood out. Based on the results of analysis, we developed a system-independent approach for automatically identifying recurrences based on symptoms, for use in environments where many users run the same software. Specifically, we proposed the comparison of stack traces and problem detection locations as a strategy for identifying recurrences. We applied this strategy using failures in two Tandem system software products and compared the results obtained with actual diagnosis and repair logs from analysts.

The comparison showed that between 75% and 95% of recurrences can be successfully identified by matching failure symptoms, such as stack traces and detection locations. Less than 10% of faults are misdiagnosed. The results show that the proposed automatic diagnosis of recurrences allows analysts to diagnose only one out of every several software failures (i.e., primarily the failures caused by new faults). In the case of a recurrence for which the underlying cause was identified, the diagnosis tool can rapidly provide a solution. In the case of a recurrence for which the underlying cause is being investigated, the diagnosis tool can prevent a repeated diagnosis by identifying previous failures caused by the same fault. These benefits are not free of cost. Misdiagnosis is harmful, because a single misdiagnosis can result in multiple additional failures. (Such a danger exists in diagnoses by analysts, also.) The proposed approach needs to be implemented in a pilot. Measurements need to be made to determine how well the approach works and to make design trade-offs.

8.4 Future Work

Future work is to continue to bring analysis of measurements and design together. An extension of this research is to address implementation issues for the diagnosis of recurrences. Given that hardware fault tolerance buys software fault tolerance, another extension is to establish empirical software fault models based on measurements and to relate the models to the design of error detection, diagnosis, and recovery strategies. An important platform for future work is massively parallel systems. At this point, little is known about software and hardware dependability issues for those systems.

Bibliography

[1] J. Gray, "A census of Tandem system availability between 1985 and 1990," *IEEE Transactions on Reliability*, vol. 39, no. 4, pp. 409–418, Oct. 1990.

[2] M. S. Sullivan and R. Chillarege, "Software defects and their impact on system availability—a study of field failures in operating systems," *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, Montreal, Canada, June 1991, pp. 2–9.

[3] A. Endres, "An analysis of errors and their causes in system programs," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 2, pp. 140-149, June 1975.

[4] T. A. Thayer, M. Lipow, and E. C. Nelson, *Software Reliability.* New York, NY: Elsevier North-Holland Publishing Company, Inc., 1978.

[5] D. M. Weiss, "Evaluating software development by error analysis: the data from the architecture research facility," *The Journal of System and Software 1*, pp. 57–70, Mar. 1979.

[6] V. R. Basili and B. T. Perricone, "Software errors and complexity: an empirical investigation," *Communications of the ACM*, vol. 22, no. 1, pp. 42–52, Jan. 1984.

[7] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification—a concept for in-process measurements," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 943–956, Nov. 1992.

[8] A. L. Goel, "Software reliability models: assumptions, limitations, and applicability," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1411–1423, Dec. 1985.

[9] J. D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application.* New York, NY: McGraw-Hill Book Company, 1987.

[10] X. Castillo and D. P. Siewiorek, "A comparable hardware/software reliability model," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh, PA, July 1981.

[11] R. K. Iyer and P. Velardi, "Hardware-related software errors: measurement and analysis," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 2, pp. 223–231, Feb. 1985.

[12] R. K. Iyer and D. J. Rossetti, "Effect of system workload on operating system reliability: a study on IBM 3081," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 12, pp. 1438–1448, Dec. 1985.

[13] M. C. Hsueh and R. K. Iyer, "A measurement-based model of software reliability in a production environment," *Proceedings of the 11th Annual International Computer Software & Applications Conference*, Tokyo, Japan, Oct. 1987, pp. 354–360.

[14] M. M. Tsao and D. P. Siewiorek, "Trend analysis on system error files," *Proceedings of the 13th International Symposium on Fault-Tolerant Computing*, Milano, Italy, June 1983, pp. 116–119.

[15] T. T. Lin and D. P. Siewiorek, "Error log analysis: statistical modeling and heuristic trend analysis," *IEEE Transactions on Reliability*, vol. 39, no. 4, pp. 419–432, Oct. 1990.

[16] R. K. Iyer, L. T. Young, and P. V. K. Iyer, "Automatic recognition of intermittent failures: an experimental study of field data," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 525–537, Apr. 1990.

[17] R. A. Maxion and D. P. Siewiorek, "Symptom based diagnosis," *Proceedings of the 1985 International Conference on Computer Design*, Port Chester, NY, Oct. 1985, pp. 294-297.

[18] J. P. Shebell, "Symptom directed diagnosis foundations and practices," *Proceedings of the 1985 International Conference on Computer Design*, Port Chester, NY, Oct. 1985, pp. 290–293.

[19] B. Latham and M. W. Swartwout, "CD_x-crash diagnostician for VMS," in *Expert Systems and Knowledge Engineering*, T. Bernold, Ed. New York, NY: Elsevier Science Publishing Company, Inc., 1986.

[20] R. A. Maxion and R. T. Olszewski, "Detection and discrimination of injected network faults," *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, Toulouse, France, June 1993, pp. 198–207.

[21] R. Chillarege, B. Ray, A. Garrigan, and D. Ruth, "The recreate problem in software failures," *Proceedings of the 4th International Symposium on Software Reliability Engineering*, Denver, Colorado, Nov. 1993.

[22] A. Avizienis and J. P. J. Kelly, "Fault tolerance by design diversity: concepts and experiments," *IEEE Computer*, pp. 67–80, Aug. 1984.

[23] B. Randell, "System structure for software fault tolerance," *IEEE Transactions on Software Engineering*, vol. SE-1, no. 1, pp. 220–232, June 1975.

[24] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell, "Fault injection for dependability validation: a methodology and some applications," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 166–182, Feb. 1990.

[25] J.-C. Laprie, "Dependability evaluation of software systems in operation," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 6, pp. 701–714, Nov. 1984.

[26] P. Velardi and R. K. Iyer, "A study of software failures and recovery in the MVS operating system," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 564–568, June 1984.

[27] J. Gray, "Why do computers stop and what can we do about it?" Tandem Computers Inc., Cupertino, CA, Tandem Technical Report 85.7, June 1985.

[28] J. Bartlett, W. Bartlett, R. Carr, D. Garcia, J. Gray, R. Horst, R. Jardine, D. Lenoski, and D. McGuire, "Fault tolerance in Tandem computer systems," Tandem Computers Inc., Cupertino, CA, Tandem Technical Report 90.5, May 1990.

[29] K. H. Kim and H. O. Welch, "Distributed execution of recovery blocks: an approach for uniform treatment of hardware and software faults in real-time applications," *IEEE Transactions on Computers*, vol. 38, no. 5, pp. 626–636, May 1989.

[30] J. H. Lala and L. S. Alger, "Hardware and software fault tolerance: a unified architectural approach," *Proceedings of the 18th International Symposium on Fault-Tolerant Computing*, Tokyo, Japan, June 1988, pp. 240–245.

[31] J.-C. Laprie, J. Arlat, C. Beounes, and K. Kanoun, "Definition and analysis of hardware- and software-fault-tolerant architectures," *IEEE Computer*, pp. 39–51, July 1990.

[32] Tandem Computers Inc., *Product Reporting System User's Guide*, June 1985.

[33] Tandem Computers Inc., *Tandem Maintenance and Diagnostic System Reference Manual*, Mar. 1989.

[34] I. Lee and R. K. Iyer, "Faults, symptoms, and software fault tolerance in the Tandem GUARDIAN90 operating system," *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, Toulouse, France, June 1993, pp. 20–29.

[35] Y. Huang and C. Kintala, "Software implemented fault tolerance: technologies and experience," *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, Toulouse, France, June 1993, pp. 2–9.

[36] Y. M. Wang, Y. Huang, and W. K. Fuchs, "Progressive retry for software error recovery in distributed systems," *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing*, Toulouse, France, June 1993, pp. 138–144.

[37] F. Cristian, "Exception handling and software fault tolerance," *IEEE Transactions on Computers*, vol. C-31, no. 6, pp. 531–540, June 1982.

[38] R. V. Hogg and E. A. Tanis, *Probability and Statistical Inference*, third edition. New York, NY: Macmillan Publishing Co., Inc., 1988.

[39] J. F. Meyer, "Performability: a retrospective and some pointers to the future," *Performance Evaluation*, vol. 14, pp. 139–156, Feb. 1992.

[40] K. S. Trivedi, J. K. Muppala, S. P. Woolet, and B. R. Haverkort, "Composite performance and dependability analysis," *Performance Evaluation*, vol. 14, pp. 197–215, Feb. 1992.

[41] I. Lee and R. K. Iyer, "Analysis of software halts in the Tandem GUARDIAN operating system," *Proceedings of the 3rd International Symposium on Software Reliability Engineering*, Research Triangle Park, NC, Oct. 1992, pp. 227–236.

[42] I. Lee, D. Tang, R. K. Iyer, and M. C. Hsueh, "Measurement-based evaluation of operating system fault tolerance," *IEEE Transactions on Reliability*, vol. 42, no. 2, pp. 238–249, June 1993.

[43] I. Lee, R. K. Iyer, and D. Tang, "Error/failure analysis using event logs from fault tolerant systems," *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, Montreal, Canada, June 1991, pp. 10–17.

[44] D. Tang, "Measurement-based dependability analysis and modeling for multicomputer systems," Ph.D. dissertation, University of Illinois, Urbana, IL, Oct. 1992.

[45] W. R. Dillon and M. Goldstein, *Multivariate Analysis.* New York, NY: John Wiley & Sons, 1984.

[46] SAS Institute Inc., *SAS User's Guide: Statistics*, Version 5 Edition, 1985.

[47] H. Spath, *Cluster Analysis Algorithms.* New York, NY: John Wiley & Sons, 1990.

[48] C. V. Ramamoothy and F. B. Bastani, "Software reliability—status and perspectives," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 354–371, July 1982.

[49] I. Lee and R. K. Iyer, "Software dependability in the Tandem GUARDIAN System," submitted to the *IEEE Transactions on Software Engineering.*

[50] M. D. Beaudry, "Performance-related reliability measures for computing systems," *IEEE Transactions on Computers*, vol. C-27, no. 6, pp. 540–547, June 1978.

[51] R. A. Sahner and K. S. Trivedi, "Reliability modeling using SHARPE," *IEEE Transactions on Reliability*, vol. R-36, no. 2, pp. 186–193, June 1987.

[52] J. B. Dugan and K. S. Trivedi, "Coverage modeling for dependability analysis of fault-tolerant systems," *IEEE Transactions on Computers*, vol. 38, no. 6, pp. 775–787, June 1989.

[53] I. Lee and R. K. Iyer, "Identifying software problems using symptoms," *Proceedings of the 24th International Symposium on Fault-Tolerant Computing*, Austin, TX, June 1994, pp. 320–329.

[54] E. N. Adams, "Optimizing preventive service of software products," *IBM Journal of Research and Development*, vol. 28, no. 1, pp. 2–14, Jan. 1984.

[55] Y. Levendel, private communication.

Vita

Inhwan Lee received the B.S. and M.S. degrees in Electrical Engineering from Seoul National University, Seoul, Korea, in 1979 and 1985, respectively. From 1979 to 1986, he was a research engineer at the Agency for Defense Development, Korea. At the University of Illinois, he was a research assistant at the Coordinated Science Laboratory from 1987 to 1994. He has served summer internships at Tandem Computers Incorporated, and at the VLSI Laboratory of Texas Instruments Incorporated. His current research interests include fault-tolerant computing, measurement, performance and dependability evaluation, reliable software design, and computer architecture. Upon completion of his dissertation, he will join Tandem Computers Incorporated, in Cupertino, California.